CAPSULES: EXPRESSING COMPOSABLE COMPUTATIONS IN A PARALLEL PROGRAMMING MODEL

A Thesis
Presented to
The Academic Faculty

by

Hasnain A. Mandviwala

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science, College of Computing

Georgia Institute of Technology
December 2008

CAPSULES: EXPRESSING COMPOSABLE COMPUTATIONS IN A PARALLEL PROGRAMMING MODEL

Approved by:

Professor Umakishore Ramachandran,
Advisor
School of Computer Science, College of Computing
Georgia Institute of Technology

Associate Professor James M. Rehg
School of Interactive Computing, College of Computing
Georgia Institute of Technology

Associate Professor Santosh Pande
School of Computer Science, College of Computing
Georgia Institute of Technology

Dr. Kathleen Knobe
Software Services Group (SSG)
Intel Incorporated

Assistant Professor Milos Prvulovic
School of Computer Science, College of Computing
Georgia Institute of Technology

Date Approved: 30 June 2008

To my beautiful wife

Lulua,

without her support, this thesis would not have been possible.

ACKNOWLEDGEMENTS

First of all I would like to thank my adviser, Professor Umakishore Ramachandran for providing me with the opportunity to get to where I am. I have had the great pleasure of working with him since I was an undergraduate at Georgia Tech and he has been a great mentor throughout all those years.

I would also like to thank my colleague and mentor, Dr. Nissim Harel, from whom I first got introduced to research. Nissim taught me much about research, without which I would not have had the many successes and accomplishments I have had throughout these short years.

I would also like to thank the members of the Embedded Pervasive Laboratory at Georgia Tech, (http://wiki.cc.gatech.edu/epl/) not only for their technical help but also their support.

I would especially like to thank Research Scientist Dr. Rajnish Kumar for providing me the emotional and motivational support in continuing my work when I had lost all hope. I enjoyed the table tennis games and discussions we had over tea breaks as they helped me re-center my efforts and produce this work.

I would also like to thank my colleagues Bikash Agarwalla, David Hilley, Junsuk Shin, Nova Ahmed, Dave Lillethun, Namgeun Jeong, Xiang Song, Dr. Arnab Paul and Dr. Sameer Adhikari for their encouragement and occasional conversations both at the technical and philosophical level. I would also like to thank other members of my lab namely, Dr. Mathew Wolenetz, Dushmanta Mohaptra and Mungyung Ryu for keeping our lab buzzing with activity.

A special mention is deserved by Matthew Flagg and Dongshin Kim, members of the *Wall Lab* who worked in providing me with computer vision applications, technical help

and valuable feedback to evaluate this work.

I would also like to mention one of my teachers, Associate Professor James M. Rehg who was the inspiration for me to pursue Computer Vision one of my few passions I still feel excited about.

I would also like to thank fellow PhD students Dr. Syed Adeel Khalid and Raffay Hamid, who have been good friends, fellow countrymen and an occasional classmate, with whom I spent many sleepless nights discussing the paradox of life, religion, and the inescapable path every PhD student must take. I especially enjoyed my days building and flying model airplanes with Adeel, which helped me take my mind away from research and all its lesser liked challenges! I would also like to mention some great friends Syed Ali Kamil, Mohammad Shaharyar Khan and Muhammad Osman Yousuf, who helped me get through the final days of my PhD. They encouraged and cheered me on when I needed the support from my friends the most.

I would also like to thank Senior Research Scientist Dr. Jean Manuel (JM) Van Thong, whom I worked with at Compaq/HP Cambridge Research Laboratory where he was not only a host, but a mentor to me. I am truly indebted to his constant encouragement to not lose hope and continue pursuing my PhD.

I would also like to thank Dr. Kathleen Knobe for exposing me to TStreams, which ultimately led me to find a topic for my thesis. She helped me, albeit unknowingly, to focus on the depth and generalization of research, a quality which I have come to value greatly in the later years.

Lastly, this work would not have been possible without the funds provided in part by NSF (NSF ITR grant CCR-01-21638, NSF NMI grant CCR-03-30639, NSF CPA grant CCR-05-41079), and the Georgia Tech Broadband Institute. The equipment used in the experimental studies is funded in part by an NSF Research Infrastructure award EIA-99-72872, and Intel Corporation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

A well-known problem in designing high-level parallel programming models and languages is the “granularity problem”, where the execution of parallel tasks that are too fine-grain incur large overheads in the parallel runtime and adversely affect the speed-up that can be achieved by parallel execution. On the other hand, tasks that are too coarse-grain create load imbalance and do not adequately utilize the parallel machine. In this work we attempt to address the issue of granularity with a concept of expressing “composable computations” within a parallel programming model called “Capsules”.

In Capsules, we provide a unifying framework that allows composition and adjustment of granularity for both data and computation over iteration space and computation space.

The Capsules model not only allows the user to express the decision on granularity of execution, but also the decision on the granularity of garbage collection (and therefore, the aggressiveness of the GC optimization), and other features that may be supported by the programming model.

We argue that this adaptability of execution granularity leads to efficient parallel execution by matching the available application concurrency to the available hardware concurrency, thereby reducing parallelization overhead. By matching, we refer to creating coarse-grain Computation Capsules that encompass multiple instances of fine-grain computation instances. In effect, creating coarse-grain computations reduces overhead by simply reducing the number of parallel computations. Reducing parallel computation instances in turn leads to: (1) Reduced synchronization cost such as that required to access and search in shared data-structures; (2) Reduced distribution and scheduling cost for parallel computation instances; and (3) Reduced book-keeping costs consisting of maintain data-structures such as blocked lists for unfulfilled data requests.

Capsules builds on our prior work, TStreams, a data-flow oriented parallel programming framework. Our results on an CMP/SMP machine using real vision applications such as the Cascade Face Detector, and the Stereo Vision Depth applications, and other synthetic applications show benefits in application performance. We use profiling to help determine optimal coarse-grain serial execution granularity, and provide empirical proof that adjusting execution granularity reduces parallelization overhead to yield maximum application performance.

CHAPTER I

INTRODUCTION

1.1 Thesis Statement

Parallel programming is difficult [76]. Even more daunting is the task of writing a parallel program that executes efficiently on hardware with varying amounts of available concurrency *without* source code modification. Different platforms provide a different level of hardware parallelism, for example, the Cell B.E. processor [31] has 1 Power Processing Element (PPE) and 8 Synergistic Processing Elements (SPEs), whereas the Intel Core2 Quad [28] processor has upto four general purpose cores. Even on a given platform, depending on the workload mix, the parallelism available for a given application may change over time. Clearly, an application programmer would like to exploit all available hardware parallelism without having to re-compile code at least when running on platforms with the same instruction set architecture (ISA). The traditional solution to this problem of adaptability has been to extract all potential application parallelism and map it evenly among available processors. However, if the granularity of parallel tasks is too fine, and the available hardware concurrency does not match the application concurrency, the application incurs excessive runtime overhead in executing these fine-grain computations. Ideally, one would like to shield the application programmer from the vagaries of resource availability while maximizing performance. Therefore, there is a need to dynamically adapt the application granularity, without changing the application source, to match the available hardware parallelism and thus reduce the *parallelization overhead*.

Current parallel programming models lack the semantic ability to express a granularity adaptation mechanism for parallel tasks, where the execution granularity could be changed for greater execution efficiency. Previous high-level parallel programming models

such as Jade [43, 69, 70], Cilk [9], OpenMP [10, 27, 42] and even surveys [5] on parallel programming trends have acknowledged the problem of high runtime overhead when executing fine-grain computations. However, the granularity problem is not addressed at the programming model level and the programmer is left to encode parallel tasks to have sufficient granularity and avoid high parallelization overheads. New models like the Intel Thread Building Blocks (TBB) [29] have automatic granularity control supported for only upto two dimensional data. However these models do not have a clearly defined semantic of a unified granularity control for both computation and data. Moreover, granularity control has also been experimented within compiler driven technologies such as in the PROMIS [71] parallelizing compiler. However, compiler driven technologies focus on an approach of data and computation decomposition into appropriate grain sizes, a technique that does not work when problem size limits are unknown.

The thesis backed by this dissertation is the following: It is possible to create a parallel programming model in which a programmer expresses the computation and data at the finest grain, and yet does not incur unnecessary parallel programming overhead at runtime. The key insight that backs this thesis and which is elaborated in the dissertation is the idea of *composable computations* that dynamically creates at runtime, coarse-grain computations and data from the fine-grain representations, commensurate with the available hardware parallelism.

1.2 Problem Statement

Current parallel programming languages and models lack the semantic ability to express computations where their granularity of execution can be altered dynamically. The ability for dynamic granularity adaptation is important because parallel applications begin to incur relatively large runtime overheads if concurrent tasks are too fine-grain and not enough hardware concurrency is available. Similarly, if computations are too coarse-grain, the

hardware concurrency is not adequately utilized leading to a load imbalance. Ideally, a parallel program should be able to automatically adapt to the available resources by adjusting its granularity and increasing its execution efficiency dynamically. To achieve this goal, we need a programming model where the programmer can write the program once, and leave the runtime system to dynamically adapt execution granularity to maximally utilize the hardware concurrency and minimize the parallelization overhead.

1.3 Contribution

In this dissertation, we address the granularity problem by investigating how efficient composable computations can be expressed within the context of a parallel programming model. We introduce the *Capsules*[1] parallel programming model, which exposes composable computations and allows the dynamic adjustment of execution granularity for concurrent tasks. We propose a unifying framework, where the programmer can compose computations over both computation space and n-dimensional iteration space. We define software abstractions that enable composability along with rules that restrict composability. It is important to note that the current Capsules runtime does not automatically determine application granularity. Application granularity may be dynamically specified or altered by the programmer or determined by a runtime profiling component that monitors the resource availability.

A Capsules application is written with the granularity that makes sense from the point of view of the application. The software abstractions in Capsules allow dynamic composition of fine-grain computations into coarser-grain modules that execute more efficiently than the corresponding fine-grain representation. Such efficient execution is possible due to the following two reasons: (1) The runtime needs to manage fewer coarse-grain parallel tasks resulting in reduced book-keeping, scheduling, and distribution overheads. (2) Fewer

[1]The term *Capsules* was coincidental also used to name another system [20] built to enable shared memory synchronization for a message passing based programming model called Concurrent C/C++. The naming of our programming model to Capsules, however, was inspired by its reference to collections of concurrent computations encapsulated into a single abstraction.

synchronization points are required to access coarser-grain shared data thereby reducing synchronization overheads in the overall execution.

Furthermore, we show that the Capsules model allows the application programmer to not only make decisions on adjusting the granularity of execution, but also allows him/her to adjust the granularity of other features. Features such as garbage collection (GC) of data can be made to occur at different granularities depending on how aggressive the programmer would like it to be. Similarly, features such as check-pointing and debugging can also occur at different granularities.

To evaluate the Capsules programming model and its runtime implementation, we parallelize three real vision applications and a synthetic application, namely: (1) The Cascade Face Detector (FD) [77], (2) the Stereo Vision Depth (SV) [83] algorithm, (3) an Apply Filters kernel used in robot path planning, and (4) a synthetic N-Stage Pipeline application. Our results show that increasing execution granularity helps reduce runtime overhead and simultaneously yields increased application performance.

CHAPTER II

CAPSULES PARALLEL PROGRAMMING MODEL

2.1 Composing Computations Dynamically

In our work, we build upon the TStreams [38] parallel programming model to incorporate the notion of Composable Computations to enable adjustable granularity. We call our new parallel programming model *Capsules*. A user of Capsules can express maximum potential application parallelism by defining an application task-graph using finest-grain computational pieces and finest-grain data abstractions. Then, fine-grain computations can be dynamically composed together by the user to form more efficient coarse-grain computations. The mechanisms for composability are divided into two sub-mechanisms that complement each other. They are: (1) *Composition over Computation Space* (Chapter 3), and (2) *Composition over Iteration Space* (Chapter 4). Each mechanism is dynamic, and allows runtime determination of granularity that enhances application performance.

2.2 TStreams

The TStreams [38] parallel programming model was developed by combining ideas from Dataflow [4], Tuple Spaces [21] and Streaming computations [53, 60] to enable a model where parallelism and data-dependencies could be cleanly expressed separately from the distribution and scheduling policies of concurrent tasks. In TStreams, the application programmer expresses all the available potential parallelism by describing fine-grain computations and data communication specification between them via an application task-graph. More details on TStreams are available in Section 10.2.2, but a brief introduction to the model is given below.

The TStreams task-graph consists of computational objects called *Step Spaces*, data

objects called *Item Spaces* and identifier objects called *Tag Spaces*. The term *space* here represent object instances that may exist during program execution. Therefore, Step Spaces represent the possible *Step Instances*, Item Spaces represent the possible *Item Instances* and Tag Spaces represent the possible unique n-dimensional identifiers, or *Tag Instances*, for the individual Step instances and Item instances. The Tag instances also act as a control mechanism in the data-flow specification of the task-graph. Therefore, every time a new Tag instance is created, new Step instances and Item instances are created in the runtime system.

The TStreams task-graph also specifies the producer and consumer relationships between the object spaces. Steps Spaces (computations) can produce into one or more Item Spaces (data) or Tag Spaces (identifiers/control). Likewise, Step Spaces can also consume from one or more Item Spaces. There is a third relationship in TStreams that is expressed between Tag Spaces and Step/Item Spaces. Each Step/Item Space is always *parametrized* by a Tag Space that uniquely identifies the Step/Item instances in the space. The *parametrize* relationship here denotes the one-to-one mapping between the Step/Item instances and the Tag instances that identify them. Therefore, each Step/Item instances in a given Step/Item space is specified by only one unique n-dimensional identifier. A Tag Space can also parametrize more than one Step/Item spaces. Therefore, each Tag instance in a Tag Space could identify Step/Item instances from different Step/Item spaces.

A sample TStreams application task-graph is illustrated in Figure 1. The oval shapes in the task-graph represent Step Spaces, the rectangular shapes represent Item Spaces and the triangles represent Tag Spaces. Note that each Tag Space is uniquely named and has its iteration dimensions defined. Furthermore, producer and consumer edges are represented by directed edges whereas the parametrize relationship is represented by the dotted line.

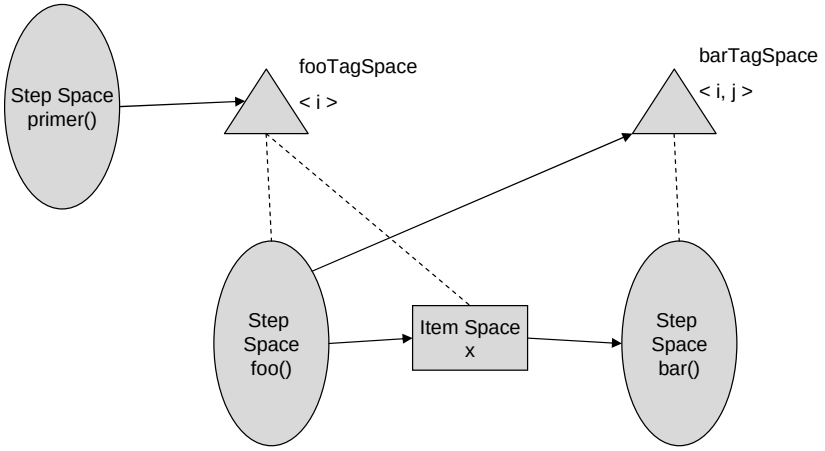


Figure 1: A TStreams application task-graph

2.3 Capsules Abstractions

In this section we describe software abstractions that allow expressing composable computation within a parallel programming model. These abstractions are (1) *StepCapsules* representing collections of computations, (2) *ItemCapsules* representing collections of data, and (3) *TagCapsules* representing a collection of n-dimensional identifiers. These abstractions are similar to the primary objects in TStreams [38] called Steps, Items and Tags, and differ only in the extra information they encapsulate to allow composability. Each Capsule object either contains only *one* object instance, or a *collection* of object instances representing a coarser-granularity. The granularity of these Capsules is user-defined, and can be dynamically determined at runtime when these Capsules are created.

To make a distinction between static and dynamic information about capsule objects, each object abstraction is separated into *Spaces* [38] and *Instances*. The notion of Capsule Spaces is analogous to the notion of *Classes* in Object Oriented Programming (OOP), which refers to the static specification of the object. Capsule Instances, therefore, are dynamic incarnations that conform to the specification of a Capsule Space (or object class). These distinctions provide Capsules with clean object oriented semantics, making it an easy development model for parallel programming.

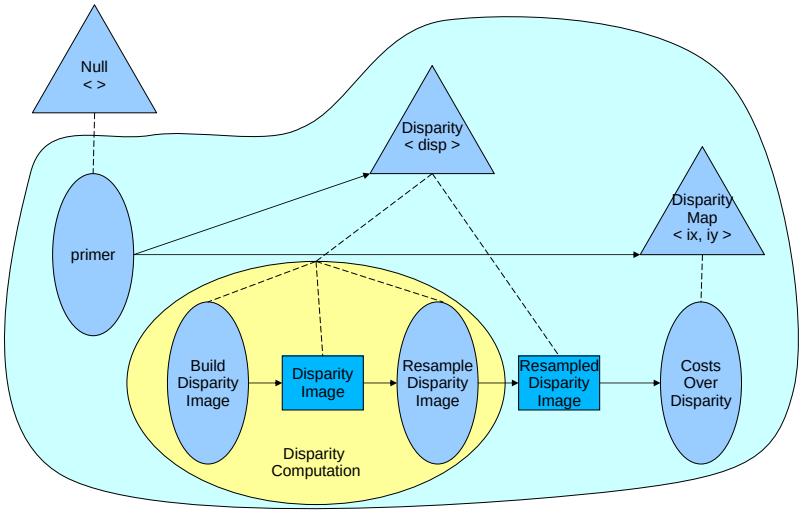


Figure 2: Stereo Vision Depth (SV) algorithm task-graph in Capsules

Figure 2 illustrates an application constructed using Capsules. The task-graph denotes Capsule Spaces and relationship edges that remain static during program execution. The triangular shapes represent TagCapsule Spaces that denote iteration spaces for computation and data. The relationship between the iteration spaces and the computation and/or data is denoted by the dotted line. The oval shapes here represent computations or StepCapsule Spaces. Finally, the rectangular shapes represent data or ItemCapsule Spaces. These store data objects communicated between computations during program execution.

2.3.1 Finest-Grain Instances

Listed below are the three basic objects found in TStreams that we use and extend from in Capsules:

Tag Instances are unique identifiers for a given Step or an Item instance (similar to Tuples in Linda [21, 11, 12]). Tags are multi-dimensional, where each dimension represents an iteration dimension specifying a range of possible values. These dimensions can be of any arbitrary type but only integer Tag dimensions are supported in the current implementation. A collection of Tag Instances are called TagCapsule Instances.

Step Instances are function calls to the finest-grain user-defined indivisible computations. Each step instance is uniquely identified by a parametrizing Tag instance. Step

instances produce Item instances or Tag instances via the producer relation. They also produce ItemCapsule instances and TagCapsule instances. Step instances also consume Item instances and ItemCapsule instances via the consumer relationship. A collection of finer-grain Step Instances is called a StepCapsule Instance.

Item Instances are fine-grain data produced by other computation Step instances. Each item instance is uniquely identified by a Tag instance. A collection of Item Instances is called an ItemCapsule Instance.

2.3.2 Capsule Instances

Now we list objects specifically added to Capsules to allow for composability:

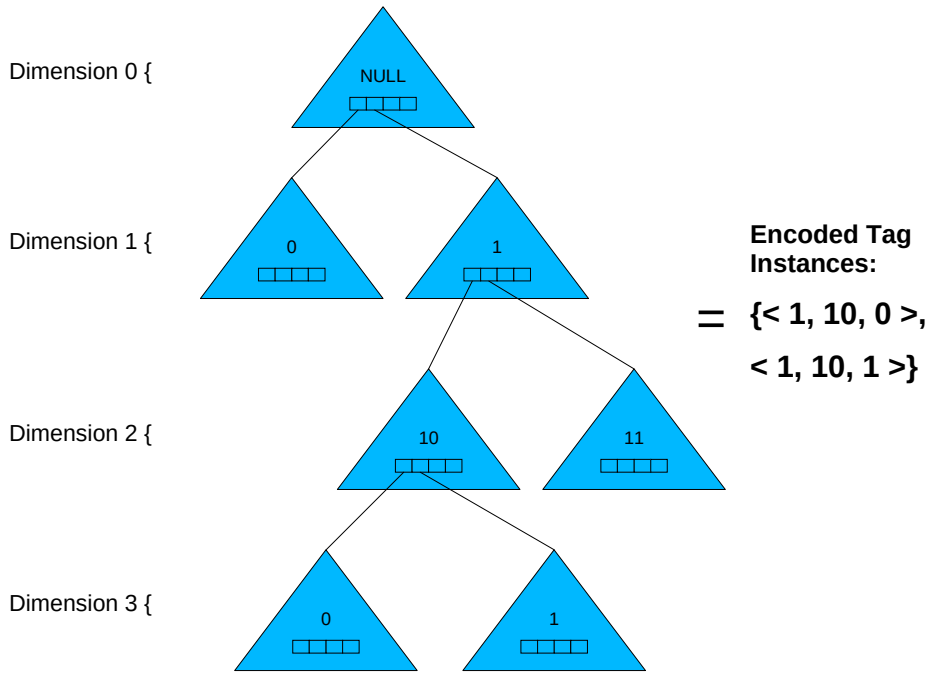


Figure 3: TagCapsule Instance. A tree encoded with specific Tag Instances for a 3 dimensional TagCapsule Space.

TagCapsules Instances (illustrated in Figure 3) are *tree* structures that store multiple Tag instances in a compressed form. The TagCapsule abstraction is the what enables composition over iteration space in the Capsules programming model. The depth i of the tree

represents the dimension i of a Tag instance. Each tree node consist of a Tag dimension value. Enumeration of Tags is achieved by the cross-product of a Tag dimension value at depth i with the child Tag dimension values at depth $i + 1$. Since trees have a hierarchical structure with fewer root nodes than child nodes, this structure also specifies the hierarchical compression of the Tag's dimension values at different dimensions. Tag dimension values that are higher in the tree are compressed more (have fewer nodes representing them) than Tag dimension values lower in the tree.

The root node of the TagCapsule instance tree represents the 0^{th} iteration dimension or the *Null dimension*. In fact, a TagCapsule tree with only the root node represents the *Null* TagCapsule instance, the only member of the *Null* TagCapsule Space. In fact, every TagCapsule Space with N defined dimensions (where $N \geq 1$) represents a cross product with the *Null* TagCapsule Space and a N dimensional TagCapsule Space. That is why every TagCapsule instance has a *Null* TagCapsule instance at its root node.

Each TagCapsule instance tree can also be uniquely identified by the *first* Tag instance from the ordered list of Tags that are contained within it. The first Tag or *Tag-key* is used as the key into operators such as *insert* and *query* into RB-Tree data-structures (Section 8.6.4) that act as containers for ItemCapsule instances.

StepCapsule Instances are coarse-grain computations that are composed from other coarse-grain Step, Item and Tag Capsules enabling composition over computation space. StepCapsules play a dual role in the composable computation paradigm. They not only represent coarse-grain computations, but also represent the GC boundary for an automatic constrained GC mechanism (described in detail in Section 3.3.3). StepCapsule instances are also hierarchical tree data-structures, where each non-leaf node represents a coarse-grain computation and a leaf node represents fine-grain Step instances.

ItemCapsule Instance is also a collection of Items forming a coarse-grain data Capsule. It is also a tree structure similar to the TagCapsule instance tree. Each node at depth i of the ItemCapsule instance tree represents the Tag dimension value of the parametrizing

TagCapsule instance tree at the same depth i. At the leaf-nodes of the tree, the actual items are stored. The items stored in a leaf-node are parametrized by the Tags represented by the parent hierarchy of the leaf-node.

2.3.3 Capsule Spaces

Finally, we enumerate the Capsule primitives that specify the static relationships in the application task-graph. These Spaces, encapsulate the common denominator properties of Capsule object instances that belong to the same space.

TagCapsule Spaces contain the static dimension information, namely, the number of dimensions in the iteration space and the name of each dimension. TagCapsule Spaces also store information about the objects they *parametrize*. Parametrization is a relationship between TagCapsule Spaces and other ItemCapsule Spaces or StepCapsule Spaces that specify which objects the TagCapsule instances uniquely identify.

StepCapsule Spaces contain static information about its parent StepCapsule Space, its parametrizing TagCapsule Space and child that are contained within it. They also contain producer/consumer relationship information between itself and other ItemCapsule and TagCapsule Spaces.

ItemCapsule Spaces also contain static information about its parametrizing TagCapsule Space and its parent StepCapsule Space.

2.3.4 Relationships between Spaces

These are the relationships or edges that exist between Space objects in Capsules.

Consumer relationship When a StepCapsule Space takes input from an ItemCapsule Space, it is called a consumer relation.

Producer relationship When a StepCapsule Space outputs to an Item/Tag Capsule Space as the result of its computation, the interaction is named a producer relationship.

Parametrize relationship The relationship between TagCapsule Instances and Item/Step Capsule Instances it uniquely identifies is called the parametrize relationship. The parametrize

relationship in capsule spaces allows definition of the iteration space and its dimensions for a computation StepCapsule Space or data ItemCapsule Space. The parametrize relationship is also an identity relationship, meaning that a given Tag Instance parametrizes an Item/Step Instance with the same Tag Instance value. Moreover, since in Capsules we have collections of Tag/Item Instances, a TagCapsule instance parametrizes a Step/Item Capsule Instance. The term *prescription* (or *prescribed*) used in TStreams is also used synonymously with the term *parametrize* (or *parametrized*).

2.4 Programming Model

In Capsules, the user specifies an application task-graph using the Capsule Space objects and relationships between them. For each uncomposed StepCapsule Space, the user specifies a *stepper* function that represents the finest-grain computation. Whenever a stepper function is called, it represents a unique Step Instance identified by a unique Tag instance. These Step instances can put/get Item instances from the Capsules runtime based on the pre-defined producer/consumer relationships in the Capsules task-graph. Step instances can also produce Tag instances to prescribe other Item/Step instances.

As described in detail in Chapter 3, Step instances can be composed together over Computation Space to form coarse-grain StepCapsule instances. Such compositions are enabled by the StepCapsule Space abstraction, that allows composing finer-grain Step/Item/Tag Capsule Spaces into coarse-grain StepCapsule Spaces. A static StepCapsule Space hierarchy is constructed by the user using the Capsules API that describes the chosen Computation Space composition. At run time, the user can then choose to execute the composition serially at any level of the StepCapsule Space hierarchy effectively creating a coarse-grain StepCapsule execution.

Similarly, Chapter 4 describes how Step/Item/Tag instances of the same type can be

composed together via Composition over Iteration Space to form coarser-grain Step/Item/-Tag Capsule instances. Such compositions over Iteration Space are done using the Tag-Capsule Space abstraction that represents the dimensionality of Steps and Items. By creating coarse-grain TagCapsule instances, the granularity of prescribed Step/Item Capsule instances is adjusted. Granularity of TagCapsules is defined over known edges of the application task-graph where *dimensional expansion* and *dimensional reduction* occur. Dimensional expansion is where new iteration dimensions are being created by a computation, whereas dimensional reduction is where existing iteration dimensions are being reduced by a computation. The user again creates coarse-grain Step/Item Capsule instances from fine-grain stepper functions at these dimensional expansion/reduction points.

The Capsules model also allows the user to create compositions over both Computation Space and Iteration Space. Such compositions allow the user to create even more coarse-grain computations spanning distinct computations and distinct iteration instances. When composing over both these dimensions, the user has the ability to serialize the coarse-grain StepCapsule instance in multiple ways via serialization schedules. The Capsules runtime can allow dynamic selection between different serialization schedules that could potentially help improve application performance.

2.5 Trade-off: Parallelism vs. Cost of achieving Parallelism

In this section we discuss the relationship between parallelism and overhead costs.

Amdahl's Law [2] states that the speed-up factor achieved by increasing N the number of processor resources made available to a given parallel program is theoretically bounded by F, the fraction of a computation that is sequential *i.e.,* computation that cannot be parallelized:

$$Maximum\ Speedup = \frac{1}{F + \frac{1-F}{N}}$$

Therefore, if you consider the limits of the equation above, it is clear that speedup would only be great if either for a fixed F, the number of available processors N is relatively

small, or for a fixed N, the application has a relatively small value of F. Applications with a small F value are also known as embarrassingly parallel applications.

If an application does have a small F value, the parallelizable serial computations can be broken down into small parallel tasks and speed-up would be great. However, in reality, breaking down a serial computation into smaller parallel tasks involves overhead, and therefore does not yield theoretical gains as predicted by Amdahl's equation. Each parallel computation incurs overheads such as data communication cost, and runtime management cost when executing in parallel. As a result, if parallel computations are too fine grain, the ratio of total overhead to the amount of useful work achieved can increase dramatically. In applications such as those explored in this work, the cost of managing fine grain parallel tasks may significantly out-weigh the benefit achieved due to increased computational parallelism. In such cases, it is important to have the right computation granularity in a parallel task to minimize overheads.

2.6 Reducing Runtime Overhead

In most parallel programming models, a large part of the overhead is incurred during *synchronization points*, which represent access to shared data either through explicit *put/get* [53, 60, 65, 59, 38] calls or implicitly through data access mechanisms such as *closures*, *continuations* [9] or *access declarations* [69]. There may also be book-keeping costs incurred to track the data requirement of tasks running concurrently. Scheduling and distribution of tasks also contributes to this overhead. Therefore, the total overhead cost in such parallel systems is directly proportional to the number of concurrent tasks that execute and the number of synchronization points required by those concurrent tasks during the entire application execution.

Therefore, in the absence of sufficient hardware concurrency, it is important to reduce this cost of parallelization. Overhead reduction can be achieved partially by reducing the total number of parallel tasks the runtime system needs to manage during the execution

of a parallel program. Reducing the total number of tasks means increasing the amount of computation each parallel task needs to encompass. We refer to this as increasing the granularity of parallel tasks. Decreasing the number of parallel tasks can also decrease the number of synchronization points required to access shared data that itself is composed to a coarser granularity. Synchronization points are reduced by moving the shared data accesses to the boundary of coarser-grain composed computations.

Our approach towards reducing the number of concurrent tasks is to create coarser-grain tasks from finer-grain tasks dynamically during parallel execution. The finer-grain computations inside the coarse-grain computation then execute serially with respect to each other. However, the coarse-grain computations still execute in parallel with respect to other coarse-grain computations. We introduce the notion of composable computations to the programming model level to enables instances of fine-grain computations to be merged together to form coarse-grain computations. Composability also helps reduce the number of synchronization points required in the total application execution. Overall, composability helps to reduce the total parallelization overhead and yields better application speedups.

2.7 Reducing Synchronization Points

In Capsules, synchronization points or data-access points to shared data structures can be reduced by creating coarser-grain data objects and coarser-grain computations. The synchronization points accessing coarse-grain data are moved to the *border* of the coarse-grain computations. Each coarse-grain computation requires a *serialization schedule* that defines the execution order of its constituent fine-grain computations. For StepCapsules created by composing over iteration space, the serialization schedule is determined by inspecting the StepCapsule instance's parametrizing TagCapsule instance (see Section 4.4). For StepCapsules created by composing over computation space, the serialization schedule requires analysis of data-dependencies between the component computations (see section 3.5).

Moving synchronization or data-access points to the border of the serialization schedule refers to the transformation required to the data-access pattern and the granularity of input ItemCapsules, such that the total number of synchronization points in the application execution are reduced. When a StepCapsule instance is composed over iteration space, moving synchronization points to the boundary of the coarse-grain StepCapsule depends on the relationship of the dimensions between the producer/consumer StepCapsule Space and its ItemCapsule Space. However, for a StepCapsule instance composed over computation space, moving synchronization points requires analysis of the producer/consumer edge information between the composed coarse-grain StepCapsule Space and its ItemCapsule Spaces.

2.8 Summary

In this chapter we laid the foundations of the basic idea of composable computations and how it helps reduce overheads such as synchronization points. In the next two chapters we explore in depth the ideas of composition over computation space and iteration space and how they are enabled within the Capsule programming model.

CHAPTER III

COMPOSITION OVER COMPUTATION SPACE

3.1 Introduction

In this chapter we elaborate on the concept of composition over computation space. We describe how this idea was motivated, and the requirements that make composition over computation space possible. These requirements include automatically determining a serialization schedule for composed computations. We describe how composition over computation space helps reduce synchronization points, one of the major causes of overheads in a parallel programming model. We then describe the StepCapsule Space abstraction as an enabler for composition over computation space within the Capsules programming model. We also formalize rules that describe valid compositions currently supported by the Capsules programming model.

3.2 Functional, Procedural Composition

Composition over Computation Space is based on the notion of combining distinct computations or distinct pieces of code to create coarse-grain computations. Furthermore, these composed computations allow further composability by combining with other computation pieces like an erector set [81]. Composability is a concept derived from functional and procedural languages where a coarse-grain function can be composed from fine-grain functions.

An example of composing computations in a common *functional programming language* such as Lisp would be as follows:

```
1   (defun foo (x)
      (baz (bar x)))
```

Clearly, the function *foo* is defined as an operation that takes a single argument *x* and returns a value that is the result of applying first *bar(x)* and then *baz(temp)*, where *temp* is a hidden stack variable holding the output of the *bar()* call. Overall, it can be said that *foo()* is *composed* from *baz()* and *bar()*. An equivalent representation of this concept in a *procedural programming language* such as C is as follows:

```
int foo(int x) {
  int y = bar(x);
  int z = baz(y);
  return z;
}
```

Here, the function *foo()* is again composed of functions *bar()* and *baz()*. Furthermore, this code demonstrates explicit declarations and uses *automatic* variables (also known as stack variables) *y* and *z*, where a copy of *z* is returned to the calling stack of *foo()*. It should be noted that these variables *y* and *z* are only visible inside *foo()* – A notion referred to as the *scope* of a function variable. Scoping is also exploited in Capsules to form an automatic GC mechanism. When ItemCapsule instances go out of scope, they are automatically GC'ed by the runtime.

3.3 StepCapsule Space abstraction

The StepCapsule Space is the software abstraction that enables composition over Computation Space in Capsules. A coarse-grain StepCapsule Space can be constructed by combining together other Step/Item/Tag Capsule Spaces. We refer to these finer-grain Spaces as *Inner Spaces*. A composed StepCapsule Space also contains information about the producer/consumer relationships between the inner spaces.

3.3.1 A Hierarchy of Composable Computations

As fine-grain StepCapsule Spaces are composed into coarse-grain StepCapsule Spaces, a hierarchical StepCapsule tree is formed. Each *finest-grain* Step, Item and Tag Capsule

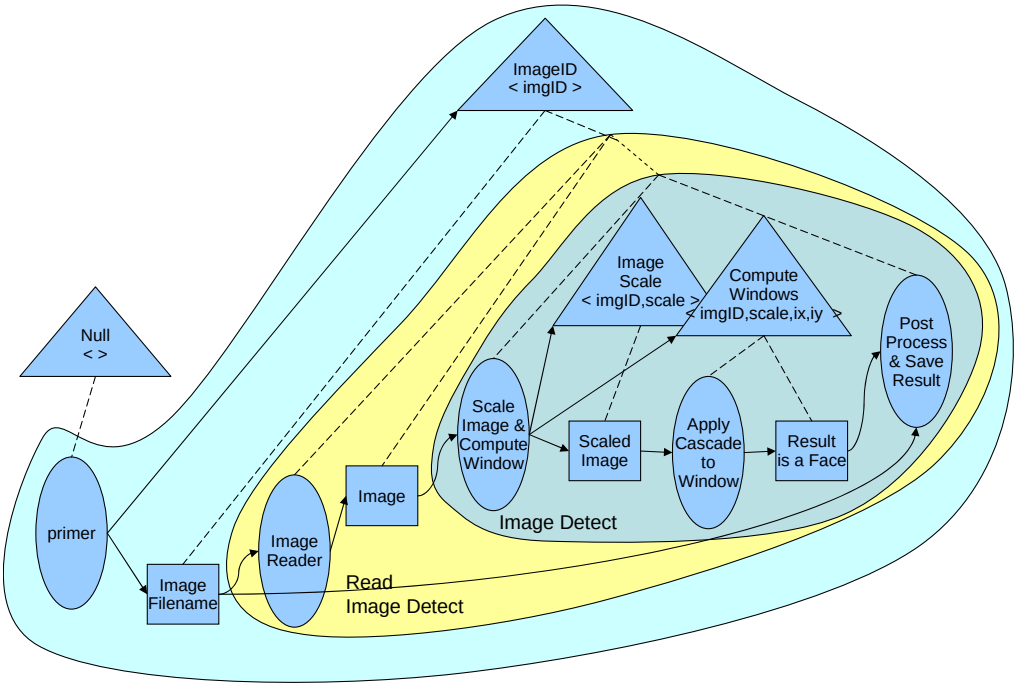


Figure 4: Cascade Face Detector Application, with a hierarchy of StepCapsule Spaces composed over computation space.

Space in the application task-graph occurs *exactly once* as a *leaf node* of this tree. Each intermediate node of the tree represents a coarse-grain *composed* StepCapsule Space. For a given application, a StepCapsule Space hierarchy tree can be constructed in multiple ways using an API. Figures 2, 4 and 36 illustrate applications in their composed hierarchical form.

Figure 4 illustrates the Cascade Face Detector Application task-graph in Capsules. Note the three hierarchical StepCapsule Space levels. The StepCapsule Space hierarchy is constructed by creating the outer-most StepCapsule Space first. The outer-most *default* StepCapsule Space represents the entire application. After creating the outer-most StepCapsule Space, the user can add inner Step/Item/Tag Capsule Spaces within it. The *primer* StepCapsule Space, the composed *Read Image Detect* StepCapsule Space, the *Image Filename* ItemCapsule Space, and the *ImageID* TagCapsule Space are all inner spaces one-level within the outer-most StepCapsule Space.

Similarly, the composed *Read Image Detect* StepCapsule Space contains inner Step/Item/-Tag Capsule Spaces. These are the fine-grain *Image Reader* and the composed *Image Detect* StepCapsule Spaces, and the *Image* ItemCapsule Space. The composed *Image Detect* StepCapsule Space in-turn contains inner spaces such as the fine-grain *Scale Image & Compute Window*, *Apply Cascade to Window* and the *Post Process & Save Result* StepCapsule Spaces, along with the *Scaled Image, Result is a face* ItemCapsule Spaces and *Image Scale* and *Compute Windows* TagCapsule Spaces.

The producer and consumer relationships into Item/Tag Capsule Spaces are expressed with respect to the finest-grain StepCapsule Space only. The relationships are expressed in a declarative manner, where the Item/Tag Capsules must be declared before being used by a producer or consumer StepCapsule Space. It is important that the Item/Tag Capsule Space be in scope (Section 3.3.3) of the producing or consuming StepCapsule Space. Scoping determines visibility of Item/Tag Capsule Spaces to StepCapsule Spaces.

The prescription relationship is also expressed in a declarative manner and a prescribed Step/Item Capsule Spaces must first define their TagCapsule Space. Note that the prescription relationship can also be inherited from the composed parent StepCapsule Space by expressing it as parent-prescribed (Section 7.2.1). The parent-prescribed parametrization simply specifies the runtime to use the parent StepCapsule Space's prescriber TagCapsule Space to identify the Iteration Space of the Step/Item/Tag Capsule Space.

3.3.2 Selecting a Computation Space Hierarchy

Constructing the right computation space hierarchy is dependent on how the application needs to be partitioned along its data and computation boundaries to extract parallelism. Selecting the best hierarchy is dependent on complex hardware and application variables. Some hardware variables are available resources such as memory and processing elements, and platform characteristics such as shared memory or distributed memory. Application

variables, on the other hand, could range from iteration space magnitude to pipeline dependencies. Furthermore, with the current trend towards hierarchical memory structures in new hardware platforms, the ability to hierarchically express computations for locality would be significant for performance. For our current system, we leave the determination of composable computation hierarchy to the application developer. In the future, perhaps we can automatically determine an optimal computation space composition hierarchy depending on the target architecture.

It is important to distinguish that the computation space hierarchy is statically defined by the user using the composition API at start-up time. The StepCapsule Space hierarchy *cannot* change once the application begins execution. However, the decision to use the composed StepCapsule Space for coarse-grain serial execution, or to execute inner StepCapsule instances in parallel, is made dynamically at runtime for each composed StepCapsule instance.

3.3.3 StepCapsule: GC Condition and Scope Boundary

The coarse-grain StepCapsule instance data-structure also acts as a GC container for all ItemCapsules contained within it. Once all inner StepCapsule instances are done executing, the coarse-grain parent StepCapsule instance is marked *executed*. The marking of a computation space composed StepCapsule instance as *executed* is also known as the *GC Condition*. Once the GC condition is satisfied, all inner data-structures (for computation and data) can be GC'ed. The StepCapsule instance is therefore also called the GC boundary for the ItemCapsules contained within it.

From a scoping perspective, ItemCapsule instances are only visible to their sibling StepCapsule instances (that have the same parent StepCapsule instance) or children of their sibling StepCapsule instances. In other words, StepCapsule spaces can see outer Item/Tag Capsule spaces defined anywhere in their parent StepCapsule Space hierarchy.

3.4 Serialization Order when Composing over Computation Space

3.4.1 Resolving Data-Dependencies

In the example above, the coarse-grain computation *foo()* was composed from fine-grain computations *bar()*, and *baz()*. However, because *bar()* and *baz()* have a data dependency between them, there is only one order in which they *can* execute, i.e. *bar()* before *baz()*. Now, consider the following example:

```
1   int foo2(int x) {              int foo3(int x) {
      int a = funcA(x);              int b = funcB(x);
3     int b = funcB(x);              int a = funcA(x);
      int c = funcC(a, b);           int c = funcC(a, b);
5     return c;                      return c;
    }                              }
```

In the example above, *foo2()* and *foo3()* are functionally identical computations that produce the same result but differ in the serialization schedule relative to functions *funcA()* and *funcB()*. These two functions are independent, and have no data dependencies between them. They represent the classical example of task parallelism, and hence is also a source of conflict that needs to be resolved when creating a serialization schedule for the coarse-grain computation *foo()*.

To execute coarse-grain StepCapsules created by composition over computation space, a serial execution schedule is therefore required to define the execution order of the fine-grain computations (Steps) contained within it. We call this the serialization order for the coarse-grain StepCapsule. In Capsules, programmers are only required to provide data-dependencies between computations with the help of producer edges and consumer edges. Therefore, in order to construct a non-blocking serial execution schedule, *a priori* resolution of data-dependencies via edge analysis is required. We perform a data-flow analysis of the application task-graph and generate a possible schedule that is free of blocking dependencies. The analysis is done only once at start-up time, and therefore has limitations in

the type of application task-graphs that are supported. The Capsules runtime currently does not support cycles in task-graphs, which eliminates the need to have a runtime data-flow analysis to be performed for every task instance during program execution. Such runtime analysis has been done in the past [7] (Section 10.2.9), of course at a higher cost that adds yet another contributing factor to the parallelization overhead.

3.4.2 Serialization Order: Iteration-Major or Computation-Major

Once a StepCapsule is composed over computation space, a question arises as to how to execute a coarse-grain StepCapsule instance when it is *also* composed over iteration space. When composed over iteration space, there are two choices for the execution serialization order for such a coarse-grain StepCapsule.

The first choice is to break the coarse-grain StepCapsule down into its respective iteration instances, and separately execute them one after another. In this case, the inner StepCapsule instances would contain only the instances belonging to the one uniquely tagged computation-space composed StepCapsule instance. Such a serialization order that maintains locality between computations is therefore called a *Computation-Major* mode of execution serialization.

The second choice is to preserve the iteration space composition and propagate it into the computation-space composed StepCapsule instance. Preserving and propagating iteration space composition allows all inner StepCapsule instances to also be composed over iteration dimensions that are common to them with their parent. Locality over composed iteration dimensions is therefore preserved for the inner child StepCapsule instances. Such a serialization order is called an *Iteration-Major* mode of execution serialization.

The Capsules programming model supports both execution modes and the decision to choose between them can be made dynamically at execution time. However, the current implementation supports the choice to be made at task-graph creation time only. A more in-depth discussion is described in a later chapter (see Chapter 5).

3.5 Synchronization Points at Capsule Boundaries

When composing over computation space, synchronization points in *get()* calls to retrieve data from external ItemCapsule Spaces are only required at the boundary of the coarse-grain computation space composed computation. Likewise, synchronization points in *put()* calls to produce data and instantiate further StepCapsule instances are only needed at the end of the computation space composed coarse-grain computation. Performing the *puts* and *gets* at the coarse-grain boundary coalesces common synchronization points and reduces the total number of synchronization points accessed during program execution. The Capsules runtime takes care of coalescing synchronization points at the boundaries of coarse-grain computations.

The *gets* at the boundary of composed computations are cached within the coarse-grain StepCapsule containers so that inner StepCapsule instances can access them. Once cached, inner StepCapsule instances within a composed StepCapsule instance are able to access the data without any synchronization. Caching therefore does not add any new synchronization points but helps reduce the overall number of synchronization points.

Similarly, *put()* calls from inner StepCapsules Spaces into outer Item/Tag Capsule Spaces from a computation space composed StepCapsule Space are cached at the composed StepCapsule Space boundary. The cached produced Item/Tag Capsules are emitted via synchronized puts once the coarse-grain composed computation has completed execution. As a result of cached and coalesced *put()* calls, the number of output synchronization points are reduced.

3.6 Rules for Constructing StepCapsule Spaces

The rules for composition over computation space are summarized as restrictions that offer the following guarantees for instances of a composed coarse-grain StepCapsule Space:

1. Execution *atomicity*

2. *Unique Tag* for each instance

3. *Termination* upon execution completion.

4. *Reachability* for all inner computation instances contained within it

The rules for composition over computation space help provide these guarantees, avoid deadlocks during parallel execution and keep the execution model simple. The rules also give execution autonomy for a composed StepCapsule instance. That is, once prescribed, the StepCapsule can be parceled off and allowed to execute anywhere without worrying about other StepCapsules. Furthermore, these rules provide fault containment at the level of a composed StepCapsule Space. That is, if a StepCapsule instance fails, the instance can be re-executed without aborting the entire computation. All of the above factors motivate us towards defining the following rules:

Rule 1: All input edges into a composed StepCapsule Space must go into inner StepCapsule Spaces.

Rule 2: All output edges from a composed StepCapsule Space must originate from inner StepCapsule Spaces.

Rules (1) and (2) imply that all inner ItemCapsule Spaces and TagCapsule Spaces, must have all their producers and consumers also inside the same StepCapsule Space. These rules also enable further composability between coarse-grain StepCapsule Spaces.

Rule 3: A StepCapsule Space is parametrized by only one TagCapsule Space.

The property in rule (3) is derived from TStreams where all computation and data spaces have to be parametrized by a Tag Space. It guarantees that all computation and data objects are uniquely tagged and derived from a single iteration space and not from multiple iteration spaces. In other words, rule (3) defines a control dependence of a StepCapsule Space computation to only one iteration space or TagCapsule Space.

Rule 4: *At least one* inner StepCapsule Space must be parametrized by the TagCapsule Space parametrizing the composed StepCapsule Space.

Rule (4), combined with rules (1) and (2), implies that all inner StepCapsules Spaces, except those specified by rule (4), are parametrized by inner TagCapsule Spaces contained within the composed StepCapsule Space. Also, these rules specify that all computations within a composed StepCapsule Space are reachable and execute at some point.

Rule 5: The composed StepCapsule Space is *logically atomic*, that is, once it begins executing, it can continue to completion without requiring further input.

Rule (5) essentially disallows cycles in Capsule task-graphs by disabling cyclic data dependence. Cyclic data dependence disables atomic execution and fault containment for StepCapsule instances, which is a crucial requirement of the Capsules parallel execution model. Rule 5 prevents inner StepCapsules Spaces from writing to outer Item Capsules Spaces and then allowing input from the same or a *derived* ItemCapsule Space back into the composed StepCapsule. An ItemCapsule Space B is said to be *derived* from an ItemCapsule Space A if B is produced by computing on Instances of A.

Figure 5 illustrates an invalid StepCapsule Space composition *LAM* that conforms to rules (1-4) but not with rule (5). The non-atomic characteristic of *LAM* is illustrated by the fact that the inner StepCapsule Space *M* requires input from an external ItemCapsule Space *Y* that is only available after a partial execution of an instance of StepCapsule *LAM*. Instances of *LAM* therefore inherently cannot execute to completion without requiring further input from the environment. Thus this composition is not a valid one under the given composition rules.

3.6.1 Checking Composition Rules

In this section we discuss how the composition rules described in Section 3.6 can be checked and whether rule checking can be automated to alleviate the burden of conformance checking for the user. Some composition rules are inherently checked in the declarative way the StepCapsule Space composition hierarchy is constructed, whereas other rules can be checked by the runtime while analyzing the task-graph. However, one rule in particular cannot be checked automatically and therefore must be adhered to by the user during application task-graph construction.

Rule (1), for example, would be checked by the declarative scoping rules of the StepCapsule hierarchy. If any composed StepCapsule Space was constructed with input edges into inner Item/Tag Capsule Spaces, *i.e.*, outer StepCapsule Spaces trying to produce into inner StepCapsule Spaces, such an edge creation would fail as the inner Item/Tag Capsule Spaces would not be *visible* to the outer producer StepCapsule Spaces.

The same is true for rule (2) as outer StepCapsule Spaces cannot consumer from out of scope inner ItemCapsule Spaces contained within other composed StepCapsule Spaces.

Similarly, rule (3) is also checked by the runtime when declaring StepCapsule Spaces.

Rule (4), however, is checked by the runtime when analyzing the task-graph for composition boundaries.

Rule (5) that disallows cycles in data dependencies is currently also checked by the runtime while detecting cycles in control dependencies during the static task-graph analysis phase. Rule (5) is easier to check automatically when cycles are completely disallowed when composing over iteration space, which is the case in this work (see Section 4.5). Cycles in composition over iteration space are disallowed by restricting computations from re-defining their own iteration dimensions. However, *atomic* execution in rule (5) would be difficult to check automatically if such restrictions were removed from composition over iteration space because it would be difficult for the runtime to detect logical cyclic data dependencies during execution. Cyclic data dependencies would cause the parallel

execution to eventually stall and would therefore be an incorrect program construction with respect to the properties of computations in the Capsules programming model. Therefore, if cycles are allowed in application task-graphs, cyclic data-dependencies would have to be checked by the user.

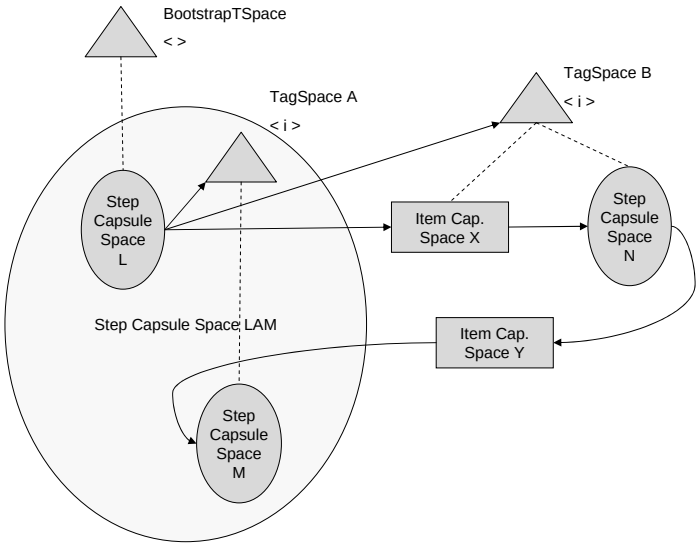


Figure 5: An example of a non-atomic (invalid) StepCapsule Space construction

3.7 Edge Relationships from Composed StepCapsule Spaces

In a later chapter (see Sections 4.7, 4.9, 4.10 and 4.11), we describe how directed edges from/to StepCapsule Spaces are classified as *fully specified input edges* (FSIE), *fully specified output edges* (FSOE), *partially specified input edges* (PSIE) or *partially specified output edges* (PSOE). Classification of input/output edges is important for the following reasons:

(1) To determine dimensional expansion and dimensional reduction points in the program because they require granularity over iteration space to be defined at such points.

(2) Edge classification is required to efficiently unpack coarse-grain data into fine-grain

Table 1: Possible input and output edge types for edges that cross computation space composed StepCapsule Spaces.

Edge type to Inner StepCapsule Space	**Edge type to Parent StepCapsule Space**
Partially Specified	Partially Specified
Fully Specified	Fully Specified/Partially Specified

data (or vice-versa while packing) during the serial execution of a coarse-grain computation.

As discussed earlier, applications in Capsules are described as StepCapsule Space hierarchies, and the user is required to specify edges to and from only the finest-grain producer/consumer StepCapsule Spaces. Since the producer/consumer StepCapsule Space and the produced/consumed ItemCapsule Spaces can be at different levels of the hierarchy, a task-graph may have edges that cross hierarchical composition boundaries. Crossing hierarchical boundaries means that finest-grain inner StepCapsule Spaces can consume from ItemCapsule Spaces that are outside the parent StepCapsule Space. The same is true for producer relationships between finest-grain StepCapsule Spaces producing into outer Item/Tag Capsule Spaces. The enumerations shown in table 1 are possible input and output edge types of a computation space composed parent StepCapsule Space given the edge type for the inner StepCapsule Space.

The edge classification for computation space composed StepCapsule spaces is important when the application chooses to execute at a coarser granularity from among the levels of the hierarchy of composed computations. Knowing the edge classification allows the runtime to determine when to retrieve the data over such edges. For example, full specification of iteration space dimensions in Fully Specified edges allows the runtime to *get* ItemCapsule instances immediately. Whereas, ItemCapsule instances over Partially Specified edges must be delayed and lazily gotton until their missing iteration dimensions can

be specified by the inner finest-grain StepCapsule instances.

The edges into and from computation space composed StepCapsule Spaces can be classified into PSIE/FSIE or PSOE/FSOE by performing a dimensional analysis between the iteration space of the composed computation and the iteration space of the Item/Tag Capsule Space (see Section 4.4.1). However, the possibilities of classifications can be narrowed down by observing the classification of edges with respect to the inner StepCapsule Spaces that consume the edge within the composed StepCapsule Space. The possible edge classification for composed parent StepCapsule Spaces with respect to the edge classification of their inner StepCapsule Spaces is as follows:

(1) Input Edges with respect to composed parent StepCapsule Space

- If an edge into an inner StepCapsule Space is a FSIE, then the edge may be either a FSIE or a PSIE with respect to the parent StepCapsule Space.
- If an edge into an inner StepCapsule Space is a PSIE, then the edge may only be a PSIE with respect to the parent StepCapsule Space.

(2) Output Edges with respect to composed parent StepCapsule Space

- If an edge from an inner StepCapsule Space is a FSOE, then the edge can either be a FSOE or a PSOE with respect to the parent StepCapsule Space.
- If an edge from an inner StepCapsule Space is a PSIE, then the edge can only be a PSOE with respect to the parent StepCapsule Space.

As described in the earlier Section 3.5, data over these merged edges is cached before serially executing the coarse-grain computations, thereby giving unsynchronized access to inner serially executing computations. The caching of edges at the coarse-grain computation boundary reduces the overall number of synchronization points required in the program's overall parallel execution.

3.8 Summary

In summary, this chapter discussed in depth the notion of composition over computation space, its motivation from functional and procedural programming and its advantages of reducing synchronization overheads. We described both natural restrictions on compositions that allow for execution correctness and temporary restrictions that create a simple model to use. We explained how the StepCapsule Space abstraction enables composition over computation space and in turn allows granularity control and features such as automatic GC. We also described mechanisms in the runtime that are required to implement composition over computation space. In particular, we described how the runtime determines serialization schedules and automatically classifies dependency edges so it can determine how to pack and unpack coarse-grain data while executing coarse-grain computations.

In the next chapter, we continue to discuss in depth the notion of composition over iteration space.

CHAPTER IV

COMPOSITION OVER ITERATION SPACE

4.1 Introduction

In this chapter we describe in depth the notion of composition over iteration space within the context of the Capsules parallel programming model.

Mathematically, Iteration Space is defined by Ramanujam et al. [62, 63] as points on a D-dimensional discrete Cartesian space. In the context of Capsules, the iteration space can be described simply as possible values that Tag instances can have. For example, a TagCapsule Space $< int\ x, int\ y >$ can span the entire space of two dimensional positive integer values. Therefore, the notion of composition over iteration space is defined by putting together a collection of Tag instances over *one or more* dimensions of a TagCapsule Space. Since Tag instances actually parametrize computations (Steps) and data (Items), composition over iteration space indirectly enables the concept of composing multiple instances of the same computation together or composing multiple instances of the same data-type together.

In the rest of the chapter, we first introduce the TagCapsule Space abstraction as the mechanism that enables composition over iteration space within the Capsules programming model. We then elaborate on how TagCapsule tree instances also define the serialization order of coarse-grain computations composed over iteration space. We also describe in detail how composition over iteration space reduces the number of synchronization points that in turn helps reduce parallelization overhead. We define the rules of composition and restriction of granularity preservation maintained by the runtime for efficient execution. We describe the edge classification performed by the runtime, which the programmer must know when composing over iteration space. Edge classification helps the runtime to know

how to handle the input/output dependencies of composed computation correctly. Lastly, we end with describing the ItemCapsule data-structure that enables composition over iteration space for data. The composition of both data and computation in Capsules gives it a unique framework to adjust granularity for both entities.

4.2 TagCapsule Space abstraction

The TagCapsule Space software abstraction enables composition over Iteration Space in Capsules. Similar to Tag Spaces in TStreams, TagCapsule Spaces in Capsules *parametrize* Item/Tag Capsules Spaces. Although more general, TagCapsules enable a behavior similar to that of tiling [62, 63, 13].

For brevity, assume the following discussion applies to capsule instances unless explicitly stated as referring to capsule spaces.

Since a TagCapsule represents a collection of Tags, when a TagCapsule parametrizes a StepCapsule, each inner Tag inherently parametrizes a Step to form a collection of parametrized Steps. However, from the stand point of the Capsules parallel programming model, all Steps parametrized by the TagCapsule are denoted as *one* coarse-grain StepCapsule that executes *atomically* and *serially* over the finer-grain Steps.

A TagCapsule denotes the same granularity for an ItemCapsule as it does for a StepCapsule it may parametrize. Therefore, this property implies that for a given TagCapsule, the parametrized coarse-grain ItemCapsule would have inner fine-grain items with a one to one mapping with the inner fine-grain Tags in the TagCapsule. For example, in Figure 6, the *fooTagSpace* parametrizes both the *foo()* StepCapsule Space and the *x* ItemCapsule Space. Therefore, the granularity of TagCapsules in *fooTagSpace* also denote the granularity of ItemCapsules in the ItemCapsule Space *x*. For example, as shown in Figure 6, the *primer* StepCapsule produces the TagCapsule $\{< 1 >, < 2 >\}$ that in turn parametrizes the StepCapsule *foo*($\{< 1 >, < 2 >\}$) and the ItemCapsule *x*$[\{< 1 >, < 2 >\}]$, both with same granularity of 2 Tags.

Furthermore, the StepCapsule Space *foo()* produces into the TagCapsule Space *barTagSpace* by expanding the one dimensional iteration space $< i >$ in *fooTagSpace* into a two dimensional iteration space $< i, j >$ in *barTagSpace*. The function *foo()* adds the dimension values $7, 8$ to the iteration dimension j to create a coarse-grain TagCapsule $\{< 1, 7 >, < 1, 8 >, < 2, 7 >, < 2, 8 >\}$.

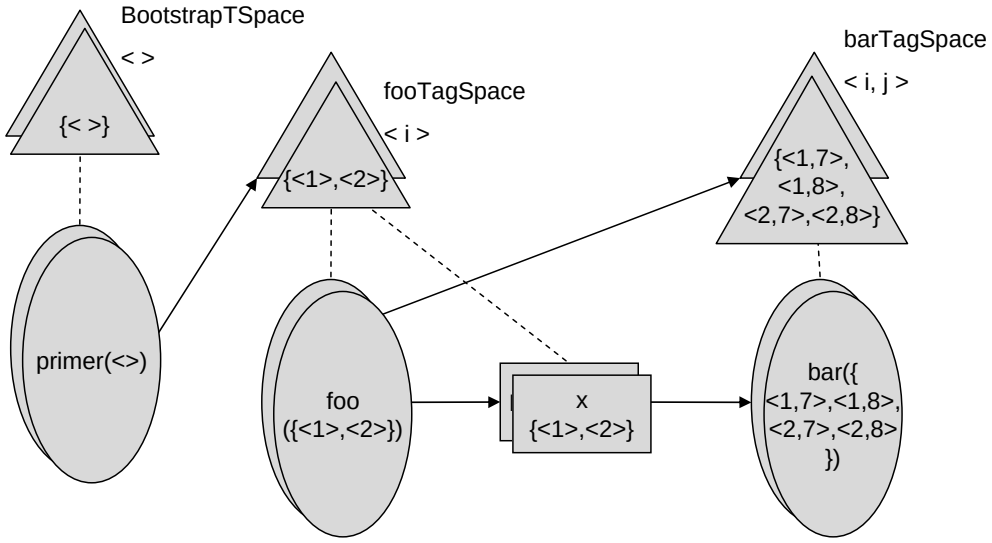


Figure 6: An example of Composition over Iteration Space

4.3 Serialization Order when Composing over Iteration Space

Similar to coarse-grain computations created by composition over computation space, coarse-grain computations created by composition over iteration space also requires a serialization schedule for execution. The serial execution schedule of a StepCapsule is based on the structure of its parametrizing TagCapsule instance tree. The schedule simply traverses the sparse TagCapsule tree in *root-left-right* order, and appends the TagCapsule instance tree value found at depth i as the value for Tag dimension i. At every leaf node at depth N, the fine-grain StepCapsule instance is executed with the enumerated Tag Instance.

Clearly, the serial execution order is dependent on the order of Tags in the TagCapsule instance tree. As Tag dimension values at any level of a TagCapsule tree are created by a

user-defined fine-grain StepCapsule function, the serialization order is therefore indirectly defined by the producers of the TagCapsules tree by the order in which they are serially produced.

4.4 Synchronization Points at Capsule Boundaries

Reducing the number of synchronization points means reducing the number of accesses to ItemCapsule Spaces. Reduce accesses to ItemCapsule Spaces in turn requires the ItemCapsule instances to also be coarse-grain so as to satisfy the data-requirements of coarse-grain StepCapsule instances.

Retrieving any ItemCapsule instance in the Capsules programming model requires the runtime to define its parametrizing TagCapsule instance. The data's parametrizing TagCapsule instance is derived from the parametrizing TagCapsule instance of the executing StepCapsule instance. Specifically, the matching dimensions between the parametrizing TagCapsule spaces of both the ItemCapsule space and the StepCapsule space determine what sub-tree in the StepCapsule instance's TagCapsule instance will be used to retrieve the ItemCapsule instance. The rules for valid producer and consumer relationships between StepCapsules and ItemCapsules are summarized in the next section. The ability to express coarse-grain ItemCapsules and performing data-access on them reduces the total number of synchronization points during program execution.

Assume the following application to illustrate the transformation that reduces synchronization points by performing them only at the coarse-grain computation boundary. The computation *foo()* is parametrized by a Tag Space with dimensions $< i, j >$ and has an input data dependency from two Item Spaces A and B, and produces an output into an Item Space C. A is parametrized by a Tag Space with dimension $< i >$, where as B is parametrized by a Tag Space with dimensions $< j >$. C on the other hand, is parametrized by a Tag Space with the same dimensions as *foo()*, namely $< i, j >$.

Therefore, the runtime has to now execute the following code in parallel for each Tag

instance in the space $< i, j >$:

```
  Ai = getItem(A, <i>);
2 Bj = getItem(B, <j>);
  Cij = foo(Ai, Bj);
4 putItem(Cij, <i, j>);
```

Here, $(I * J)$ instances of *foo()* execute in its finest granularity, where each instance requires *one* synchronized *get()* on each Item Space *A* and *B*, and *one* synchronized *put()* on Item Space *C* to satisfy its input/output data requirements. Therefore, overall $2(I * J)$ *get()* calls and $(I * J)$ *put()* calls are performed, bringing the total number of synchronization points to $3(I * J)$. However, if *foo()* is composed along the dimension i of its parametrizing iteration space, with all values of i grouped together into a single collection, only J coarse-grain instances of *foo()* would then exist. Each parallel execution of the coarse-grain instance of *foo*$< j >$ would then perform only one synchronized *get()* on each $A[0 : I]$ and $B[j]$, bringing the total to $(2 * J)$ synchronized *get()* calls. Similarly, each coarse-grain instance of *foo*$< j >$ would perform one *put()* of $C[0 : I, j]$, bringing the total number of synchronized *puts()* to (J). Overall, the number of synchronization points required during the parallel execution of all instances *foo()* over the iteration space $< i, j >$ is now only $(3 * J)$. Such a transformation reduces the number of synchronization points required for *foo*$< i, j >$ by creating coarse-grain accesses to A at the boundary of dimension i during the serial executions of *foo*$< i, j >$. We give below the transformed code executed by the runtime in parallel with coarse-grain instances of *foo()* and coarse-grain synchronized *gets()* and *puts()*.

```
  Aall = getItem(A, <0:I>);
2 Bj   = getItem(B, <j>);
  for(int i = 0; i <= I; i++) {
4   Cj[i] = foo(Aall[i], Bj);
  }
6 putItem(Cj[0:I], <0:I, j>);
```

4.4.1 Computing Dimensional Boundary

To access coarse-grain data for a coarse-grain computation the *Dimensional Boundary* between the StepCapsule Space and the ItemCapsule Space must first be determined. To be more specific, StepCapsule instances that have been composed over iteration space, access coarse-grain ItemCapsules that have also been composed over iteration space at the dimensional boundary of the StepCapsule to reduce synchronization points.

The Dimensional Boundary for input edges contain consumer edge information derived after analyzing the dimensional relationships between the ItemCapsuleSpace and the StepCapsuleSpace. The Dimensional Boundary consists of edges separated into G sets, where G is the dimensionality of the StepCapsuleSpace. These edge sets are used during the serial execution to determine the synchronization boundaries at which coarse-grain ItemCapsule instances are retrieved, and to determine unpacking points, which are used to get access to fine-grain data from within these coarse-grain ItemCapsule instances.

The Dimensional Boundary edge sets for input and output data-access can be divided into three categories. These categories are the (1) *First Dimension Dependency*, the (2) *Intermediate Dimension Dependency*, and the (3) *Last Dimension Dependency*. An edge E_k is dependent on a dimension D_i^S of a StepCapsule Space S if and only if D_i^S matches the dimension D_j^I of some ItemCapsule Space I. For example, consider an ItemCapsule Space X with dimension $< j >$ consumed by a StepCapsuleSpace *foo()* with dimensions $< i, j, k >$. For this consuming edge E_k, dimension $D_{j=1}^I$ of X, matches dimension $D_{i=2}^S$ of *foo()*. Therefore, E_k is part of set $G_{i=2}$.

The *First Dimension Dependency* variable contains an edge E_k in set G_i if and only if the first dimension $D_{j=1}^I$, matches D_i^S. Edges in this set collection define the boundary *gets()* performed during serialized execution of iteration-space composed computation. The *Intermediate Dimension Dependency* variable contains an edge E_k in set G_i if and only if an intermediate dimension within $D_{1<j<J}^I$ matches D_i^S. Edges in this set collection are used to determine the ItemCapsule trees that need to be traversed along with the StepCapsule

instance's TagCapsule tree for serial execution. The *Last dimension dependency* variables contains an edge E_k in set G_i if and only if the last dimension within $D^I_{j=J}$ matches D^S_i. Edges in this set collection are used to unpack the data from the ItemCapsule nodes at dimension $D^I_{j=J}$.

These dimension dependency edge sets help determine the dimensional boundary where coarse-grain input synchronization points are performed.

The Dimensional Boundary for data output is trivial as output is performed only at the end of a iteration-space composed computation execution. As we describe shortly (Sections 4.5, 4.7, 4.10), computations can only produce into Item/Tag Capsule Spaces that create *Dimensional Preservation* or Dimensional Expansion. Since output spaces have at least equal or more dimensions than the producing computation, all output edges are members of all three component boundary sets described above.

4.5 Rules for Composition over Iteration Space

In this section, we discuss rules that define composition over iteration space. In general, these rules provide restrictions on certain application task-graphs that make composability either expensive or impossible. We eliminate this class of application task-graphs to maintain a balance between a simple and efficient runtime and a parallel programming model that is general enough to sufficiently address the composability requirements for the class of applications targeted in this work.

To elaborate further, these rules describe restrictions on the *dimensions* of Step/Item/-Tag Capsule Spaces joined together by producer/consumer relationships. In order words, these rules define what producer/consumer edges are allowed between StepCapsule Spaces and other Tag/Item Capsule Spaces.

Since every Step/Item Capsule Space is parametrized by *only* one TagCapsule Space that defines its iteration space and dimensions, we refer to the dimensions defined by a TagCapsule Space as the dimensions of its parametrizing Step/Item Capsule Spaces.

4.5.1 Rules for output edges from StepCapsule Spaces:

Rule 1: If a StepCapsule Space has a list of N dimensions, its output Spaces must also have at least the *same* N dimensions (or more).

Rule (1) allows a StepCapsule Space to produce into an Item/Tag Space with more dimensions than itself. In other words, rule (1) allows *Dimensional Expansion* and *Dimensional Preservation* to occur but disallows dimensional reduction between a *producer StepCapsule Space* and its *recipient* Item/Tag Capsule Spaces. Restricting dimensional reduction in producer edges is necessary because otherwise a StepCapsule Space with N dimensions would create value collisions in any space that has less than the exact same N dimensions. Value collisions are invalid in the underlying Dynamic Single Assignment (DSA) [56] property in the TStreams programming model [38] that Capsules is based on.

Rule 2: The matching dimensions described in rule (1) must be in the same *contiguous* order.

Rule (2) is expressed for efficiency and is required because of the tree structure of a TagCapsule instance. The requirement for having contiguously ordered matching dimensions alleviates the runtime from re-ordering matching dimensions and provides fast querying into a TagCapsule instance tree to retrieve its Tag-key value. The Tag-key value is the first Tag instance from the ordered list of Tag instances contained within a TagCapsule instance. Therefore, Tag-keys uniquely identifying a Step/Item/Tag Capsule instance. Tag-keys are also required to perform *put()/get()* operations on ItemCapsule Spaces.

As we described earlier in Section 2.3, a TagCapsule instance is a partial tree of maximum depth N, where the value of a node at tree level N_i represents a value of the N^{th} dimension of a Tag. Each sub-tree within a TagCapsule tree represents values for the corresponding dimensions. Since output and input edges are dependent on any contiguous subset of dimensions of a StepCapsule Space (rules (2) and (4)), the runtime can easily identify the sub-tree representing the dependent dimensions. Identifying these sub-trees is required before Tag-keys can be extracted from them, that in-tun are used to identify the

coarse-grain ItemCapsule instances during *put()* and *get()* operations. Therefore, rule (2) and (4) are essential to efficiently produce/consume ItemCapsules instances.

Rule 3: New dimensions on output Spaces that do not appear on the producer StepCapsule Space must be listed after the matching dimensions.

Rule (3) is required to keep the runtime light-weight and efficient by alleviating the need to re-order dimensions during dimensional expansion. Defining the new dimension at the end of the matching dimension list N, makes creating new TagCapsule instance trees an easy operation. Adding new dimension values to an existing TagCapsule tree instance requires only attaching the new value nodes as child nodes to the leaf nodes of a *clone* of the original TagCapsule instance tree.

4.5.2 Rules for input edges into StepCapsule Spaces

Rule 4: All matching dimensions between input Spaces and StepCapsule Spaces must have the same *contiguous* order.

Rule (4) is motivated with the same efficiency reasons expressed for rule (3).

Rule 5: Missing dimensions on input Spaces with respect to the consuming StepCapsule Space require *dimension definition functions*. These dimension definition functions define the values for the missing dimensions.

Rule (5) allows the expression of dimensional reductions within an application task-graph (see Section 4.9). Dimensional reduction is a property of an edge within an application task-graph where the consuming StepCapsule Space does not contain all the dimensions of the ItemCapsule Space it is consuming from. The missing dimensions can be determined by one of two ways: (1) If the missing dimensions are *non-data-dependent* they can be statically defined. (2) If the missing dimensions are *data-dependent*, they require inspection of ItemCapsule instances that are parametrized by the non-missing dimensions.

4.5.3 Rules to Simplify the Capsules Runtime

The following restrictions are created to simplify the runtime implementation. These restrictions can be removed to generalize the model even further, but would require a more complex and more efficient implementation to reduce runtime overheads.

Rule 6: Data-dependent missing dimensions can only be determined from ItemCapsule Spaces that have pre-determined dimensions, *i.e.,* , data-dependent missing dimensions cannot be dependent on data that itself has missing dimensions requiring definition.

Rule 7: There can be no cycles in a task-graph.

Both data dependent and control dependent cycles are disallowed because the *put()* API does not enable a StepCapsule to emit re-defined values on matching dimensions that parametrized the StepCapsule as well. These matching dimensions are assumed to have a one-to-one exact value mapping. Only new dimensions are allowed to be defined by the *put()* API during dimensional expansion. Such a restriction is purely for performance reasons and not due to any inherent limitation of the composability concept. Allowing StepCapsule Spaces to re-define and emit common dimensions would require the runtime to incur some additional overhead to maintain consistency while building a compressed TagCapsule tree instance for output. As we show later, a compressed TagCapsule tree representation is essential for an efficient mechanism that reduces the number of input and output synchronization points to shared ItemCapsule Spaces. The restriction on cycles can be removed by adding more mechanisms in the runtime. However, this is left for future work and is discussed in detail in Section 11.2.1.

4.5.4 Checking Composition Rules

Similar to rules for composition over computation space, the runtime can also check conformance to the rules of composition over iteration space. All rules except rule 6 are

checked during the edge analysis phase before program startup, at which time the dimensional boundaries required for coalescing synchronization points are also computed (see Section 4.4.1). Rule (6), which is a restriction on the possible data dependencies between input edges of the application task-graph, must be followed by the user when constructing task-graphs. Rule 6 allows data over only *Fully Specified Input Edges* (FSIEs) to be gotton by the runtime to help determine data-dependent missing dimensions. Access to *Partially Specified Input Edges* (PSIEs) is not allowed when determining data-dependent missing dimensions and is prohibited at runtime when performing data access *gets()* calls.

An overview of rules for both composition over computation space and iteration space is provided in Table 2.

Table 2: Overview of rules for composition over computation space and iteration space.

Rules	Motivation	Checked by	If failure to adhere
Composition over Computation Space			
1,2	Ability to compose with other computations	declarative scoping	startup termination
3	Control dependence to only one iteration space	declarative scoping	startup termination
4	Reachability of all inner computations	task-graph analysis	startup termination
5	Eliminate cyclic data-dependence	task-graph analysis	startup termination
5	if cycles enabled over iter. space in future	user	stalled execution
Composition over Iteration Space			
1	Allow only dimensional expansion/preservation	task-graph analysis	startup termination
2,3,4	Good prog. practice → efficient runtime	task-graph analysis	startup termination
5	Dimensional Reduction needs grain definition	task-graph analysis	startup termination
6	Good prog. practice → efficient runtime	user	runtime termination
7	Simple efficient runtime	task-graph analysis	startup termination

4.6 Granularity Preservation

One of the crucial assumptions for keeping the Capsules programming model simple and the runtime efficient is that of *Granularity Preservation.* Granularity preservation is defined as maintaining a collection of computation or data instances as they were initially created. That is, once a subset of instances are grouped together into a Capsule of a certain size (the capsule size is the granularity for those instances), then that collection of instances cannot be broken or merged with another for the duration of their existence.

The notion of granularity preservation is used in Capsules for several reasons. One, it simplifies the runtime mechanisms required to enable dynamic granularity control. If for example granularity preservation were not maintained, it would require further complex mechanisms to maintain knowledge of past granularity values and a mapping of instances contained within them.

Lets take the case of retrieving data to satisfy a computation's input dependencies as an example. Recall that ItemCapsules instances are collections of Items stored in a structured tree. Each collection of Items is uniquely identified by a Tag-key, which is the Tag identifying the first Item from the ordered list of items within the Capsule. So to retrieve the ItemCapsule, its Tag-key must be known.

If at any point in time the original data collection is broken up into two collections, two Tag-keys would now represent the two collections. In order to get any item from either sub-collection, the right Tag-key would have to be known. Moreover, to retrieve all items from the original collection, both Tag-keys would have to be known. The knowledge about old collections, their constituent members, their respective keys, their transformation operation (break or merge) into new collections, and their respective keys, would have to be maintained by the runtime for every collection that could possible exist and change its granularity. Such maintenance of meta data would be required to guarantee correctness and termination for the highly dynamic environment.

4.6.1 Granularity Preservation when Composing over Computation Space

Granularity preservation when composing over computation space means that once a computation space composed StepCapsule instance is serialized at a given hierarchy level, then that entire StepCapsule instance must execute atomically to completion. All inner StepCapsule instances contained within the serialized StepCapsule instance must execute serially with respect to other inner StepCapsule instances. Such a semantic restriction again creates a simple runtime and avoids having to deal with dynamic granularity adjustment cases

where the inner StepCapsules could potentially start executing again in parallel.

4.6.2 Granularity Preservation when Composing over Iteration Space

Granularity preservation when composing over iteration space refers to computation and data composed over iteration space. As described earlier, when computations and data instances are composed together into a capsule, those instances must remain members of the same capsule and not split into smaller capsules or merge with other capsules to form larger capsules. Such a restriction again keeps the runtime simple without having to manage relationships of capsules and their members that would otherwise be required for a system that did not have such a property.

4.7 Dimensional Expansion in Output Edges

The composition and granularity of an Item/Tag Capsule instance is dynamically controlled by the user-defined stepper function that perform Dimensional Expansion (see Rule 1 Section 4.5.1) on *Partially Specified Output Edges* (PSOE). These edges are called partially specified because the dimensions of the computation cannot completely specify all the dimensions of the output Item/Tag Capsule instance. These missing dimensions must therefore be defined over PSOEs, providing an opportunity to the programmer to also define the granularity over those dimensions. Automatic granularity determination is currently not supported and is left for future work (see Sections 11.2.2 and 11.2.3).

4.7.1 Identifying Dimensional Expansion Points

Dimensional Expansion is created by any finest-grain StepCapsule Space that introduces a new computational dimension (also known as the Tag dimension) into the Capsules program description. For example, in Figure 6 the StepCapsule Space *primer()* creates dimension i and the StepCapsule Space *foo()* creates dimension j. Therefore *primer()* is the dimensional expansion point for the dimension i and *foo()* is the dimensional expansion point for the dimension j.

StepCapsule Spaces that introduce dimensional expansion are the only computations able to make decisions on the granularity of a given dimension. Furthermore, StepCapsules spaces cannot participate in defining the granularity of dimensions that they do not create. Specifically, StepCapsule spaces that are parametrized by a TagCapsule Space containing the dimension i, and are therefore *not* creators of i, must maintain (or propagate) the granularity of i when producing more Tag/Item Capsule instances that reference the same dimension i. For example, *foo()*, is not a producer of i and must propagate the granularity of i in the output TagCapsule Space *barTSpace* that also references the dimension i. However, as *foo()* is the Dimensional Expansion Point for dimension j, it does have control over deciding the granularity of dimension j.

4.8 Sliced/Unsliced Dimensional Expansion

Slicing is one of the operations used during dimensional expansion that enables granularity sets from previously defined dimensions to be combined with granularity sets on newly expanded dimensions. The alternative to slicing is to have an *unsliced* dimensional expansion, where only the granularity sets of the newly expanded dimensions define the granularity of a new iteration space.

Specifically, dimensional expansion is the expansion of an iteration space of dimensionality D with predefined granularity sets into an iteration space of higher dimensionality $D+N$. During this process, dimension expansion creates new granularity sets for the added N dimensions, while at the same time preserving the original granularity sets for the existing D dimensions. The granularity of the original D dimensions has to be preserved due to the granularity preservation property maintained by the runtime (Section 4.6). Slicing therefore creates a composition on both $D + N$ dimensions whereas an unsliced expansion creates compositions on only the N dimensions.

To illustrate the concept of Sliced/Unsliced dimensional expansion, consider the task-graph shown in Figure 6 wherein a computation *foo()* iterates over a one dimensional iteration space $< i >$. We can see that *foo()* performs a dimensional expansion into a TagCapsule Space *barTagSpace* with a two dimensional iteration space $< i, j >$. Here *foo()* is the creator of the dimension j and must therefore define granularity sets for the dimension values of j. However, granularity sets for the dimension i in *fooTagSpace* must be preserved as they were defined earlier in the program by the *primer* StepCapsule Space. In other words, TagCapsule instances in *barTagSpace* must have the same granularity sets for values of i as in *fooTagSpace* and the StepCapsule instance *foo()*. However, as dimensional expansion is done at *foo()*, the granularity sets for values of i are combined with new granularity sets for values of j.

When slicing is enabled, each value for i is appended with a granularity set of values of j. For example, when executing *foo()* with granularity set values of i, $\{< 1 >, < 2 >\}$, the granularity set values $\{< 7 >, < 8 >\}$ are emitted for each i value. Then all nodes representing the dimension i values $\{< 1 >, < 2 >\}$ in the TagCapsule tree would have a copy of the values $\{< 7 >, < 8 >\}$ added to them. In other words, the emitted TagCapsule instance in *barTagSpace* would contain two dimensional Tag values that are the cross product of $\{< 1 >, < 2 >\}$ for dimension i with $\{< 7 >, < 8 >\}$ for dimension j, namely $\{< 1, 7 >, < 1, 8 >, < 2, 7 >, < 2, 8 >\}$. The example given is illustrated in the TagCapsule tree shown in Figure 7. However, if slicing is disabled, a separate TagCapsule tree is emitted for each unique value of i and a unique granularity set of j. The entire granularity set to which the value of i belongs to, is redundantly added to the TagCapsule tree. Such redundancy is required for granularity preservation and to allow the prescribing computations of *barTagSpace* to know the granularity set of i. The TagCapsule trees created with slicing disabled is illustrated in Figure 8.

To summarize, sliced dimensional expansion creates compositions over iteration space spanning multiple dimensions, and therefore creates an expanded iteration space that is

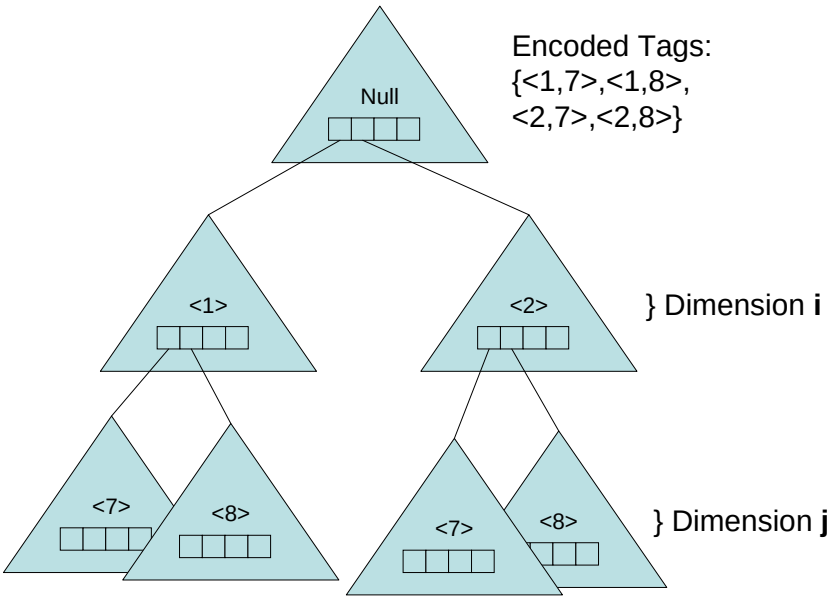


Figure 7: A Sliced two dimensional TagCapsule instance tree

coarser-grain. Unsliced dimensional expansion, on the other hand, creates compositions over iteration space on only the newly expanded dimensions. Unsliced dimensional expansion is therefore of lesser-granularity than a sliced dimensional expansion. The choice between slicing or unsliced dimensional expansion is available to the user and depends on how coarse the granularity for Item/Tag Capsules needs to be. Sliced Item/Tag Capsule instances are coarser-grain, and therefore create fewer Step/Item/Tag Capsule instances thus reducing the total parallelization overhead. Unsliced Item/Tag Capsule instances on the other hand, are of lesser granularity resulting in more instances of Step/Item/Tag Capsule instances and an increase in the total parallelization overhead.

4.9 Dimensional Reduction at Input Edges

Dimensional Reduction, also described in rule (5) of Section 4.5, occurs when a Step-Capsule Space cannot define all the dimensions of the ItemCapsule Space it is consuming from. It is similar to the reduction operation where higher dimensional data is converted into lower dimensional data. For example, a one dimensional array $X[i]$ can be reduced

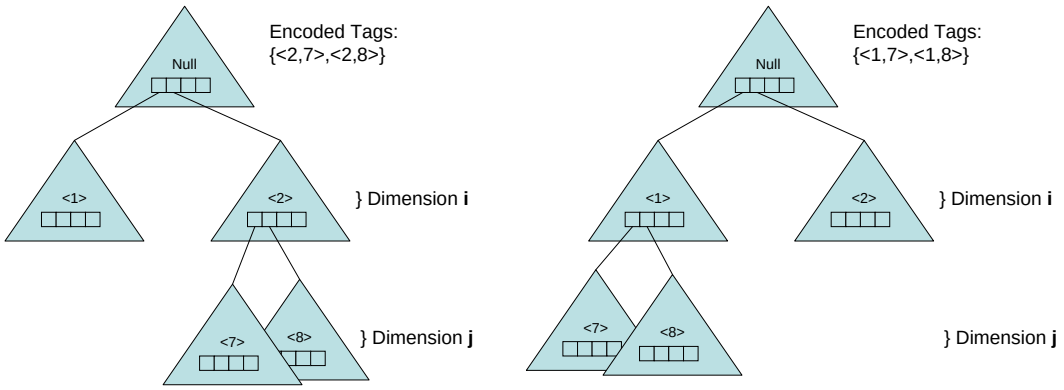


Figure 8: Unsliced Two dimensional TagCapsule instance trees representing an equivalent iteration space as the coarser-grain Sliced TagCapsule instance tree in Figure 7.

by the summation operator into a scalar value $Y = \sum_{i=0}^{i=I} X[i]$. In dimensional reduction within Capsules, one or more dimensions of the ItemCapsule Space can be reduced. For example, a StepCapsule Space *foo()* with dimensions $< i >$ consuming from an ItemCapsule Space *X* with dimensions $< i, j, k >$. Here, the edge from *X* to *foo()* is classified as a *Partially Specified Input Edge* (PSIE). An illustration of this example is given in Figure 9.

4.9.1 Sliced/Unsliced input to PSIEs

Similar to the slicing option when performing dimensional expansion in a Partially Specified Output Edge (PSOE), Partially Specified Input Edges (PSIE) also must consider whether it needs to perform slicing on any of the missing dimensions that are being defined by its dimension definition functions. If any ItemCapsule Space dimensions that are being defined by the PSIE had slicing enabled, the consuming PSIE must know of this property. Knowledge of enabled slicing is required at PSIEs because sliced dimensions create coarser-grain ItemCapsule instances and therefore fewer *gets()* need to be performed to consume all ItemCapsule instances when performing a dimensional reduction. The opposite is true for unsliced dimensions, that create finer-grain compositions and therefore require more *gets()* to retrieve all data instances to perform a dimensional reduction.

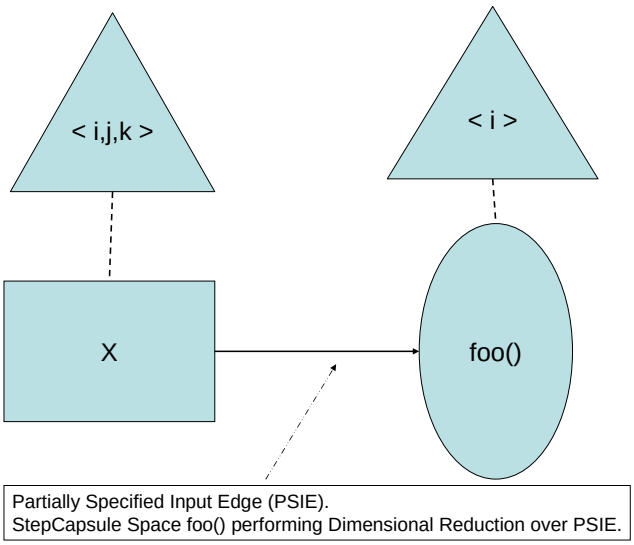


Figure 9: Example of dimensional reduction in a Capsules sub-graph.

4.10 Dimensional Preservation on Output Edges

All output edges in a Capsules application task-graph that are not partially specified are classified as *Fully Specified Output Edges* (FSOE). As the name suggests, FSOEs link computations with Item/Tag spaces that have dimensions completely specified by the dimensions of the computation itself. Moreover, since no new dimensions are being created on these edges, emitted Item/Tag Capsules have the same granularity on matching dimensions as the producing computation. Therefore, it is at FSOEs that granularity preservation is maintained. The granularity on matching dimension is preserved due to the granularity preservation restriction of the system. Since only computations with edges that create dimensions (Dimensional Expansion Points) can define granularity over an iteration dimension, all other computations that iterate over the same already created iteration dimensions must therefore preserve the granularity of such iteration dimensions.

4.11 Dimensional Preservation on Input Edges

Similar to *Fully Specified Output Edges* (FSOE), input edges that are not partially specified are classified as *Fully Specified Input Edges* (FSIE). FSIEs connect computations with

Item/Tag Capsule Spaces with dimensions that can be completely specified by the dimensions of the consuming computation. It is at FSIE where dimensional preservation is also maintained.

To illustrate an example of a FSIE, consider a StepCapsule Space *foo()* with dimensions $< i, j >$ consuming from two ItemCapsule Spaces *X* and *Y* with dimensions $< i >$ and $< j >$ respectively. Each input edge from *X* and *Y* into *foo()* is therefore classified as a FSIE as the dimensions of *foo()* can define the values of the dimensions of *X* and *Y*.

It is interesting to note a difference between FSIEs and FSOEs. FSOEs connect computations and Item/Tag Capsule Spaces that have the exact same dimensionality. However, FSIEs connect computations and Item/Tag Capsule Spaces where the dimensionality of the Item/Tag Capsule Space's matching dimensions can be less than or equal to the dimensionality of the consuming computation.

FSIEs enable the runtime to automatically unpack data received over them before executing a StepCapsule instance. In case the StepCapsule instance represents a finest-grain StepCapsule Space with a user-defined stepper function, data over FSIEs can be easily accessed into the stepper function.

4.12 ItemCapsule Spaces: Composed over Iteration Space

When creating coarse-grain computations by composing over iteration space, it is crucial to also have the ability to change the granularity of data objects. We call these composable data objects ItemCapsule instances in the context of the Capsules programming model. The granularity of ItemCapsule instances depends on the granularity of the TagCapsule instances that parametrize them. As described earlier in Section 2.3, ItemCapsules are tree data-structures that mimic the structure of their parametrizing TagCapsule instance. The similarity is data-structures is essential to allow efficient querying of the ItemCapsule tree for relevant Items that are required to fulfill the data request of the StepCapsule instances.

4.13 Summary

In this chapter, issues relating to composition over iteration space were discussed in depth. We identify two sets of points in the application task-graph where granularity over iteration space has to be defined, namely at dimensional expansion points and dimensional reduction points. We define rules that allow the definition of granularity at only such points in the program, rules that are mostly checked by the runtime. At all other points in the task-graph (FSIEs and FSOEs), the property of granularity preservation is maintained for performance and simplicity in implementing the runtime. We also illustrate details on how composition over iteration space helps reduce synchronization points at dimensional boundaries of coarse-grain StepCapsules, which in turn helps reduce the parallelization overhead.

In the next chapter we discuss issues relating to composition over both computation space and iteration space.

CHAPTER V

INTERACTION BETWEEN THE TWO COMPOSITIONS

5.1 Introduction

In this chapter we discuss in depth issues that deal with compositions over both Computation Space and Iteration Space.

Figure 10 illustrates a sample Capsules application task-graph where both composition over computation space and iteration space take place simultaneously. The task-graph contains two computations *foo()* and *bar()* that are composed together (composition over computation space) to form the coarse-grain StepCapsule Space *foobar()*. Also, *foo()* and *bar()* are prescribed by the same TagCapsule Space *TS* that iterates over a one dimensional iteration space $< i >$. When a runtime composition over iteration space is initiated by gathering values of i to form the TagCapsule instance $\{< 1 >, < 2 >, < 3 >\}$, they are emitted into the TagCapsule Space *TS*. The coarse-grain TagCapsule instance in turn parametrizes a coarse-grain StepCapsule instance *foobar(*$\{< 1 >, < 2 >, < 3 >\}$*)*. If the *foobar(*$\{< 1 >, < 2 >, < 3 >\}$*)* StepCapsule instance is to be serially executed, there are two available options for a serial execution schedule. The serialization options are illustrated in Figure 11.

The execution schedule on the left in Figure 11 keeps locality between the instances of iteration space composition and is therefore called an *Iteration-Major serialization* schedule. On the other hand, the execution schedule on the right keeps locality between the instances of the computation space composition or locality between the StepCapsule Spaces *foo()* and *bar()*. Such a serial execution order is called a *Computation-Major serialization*.

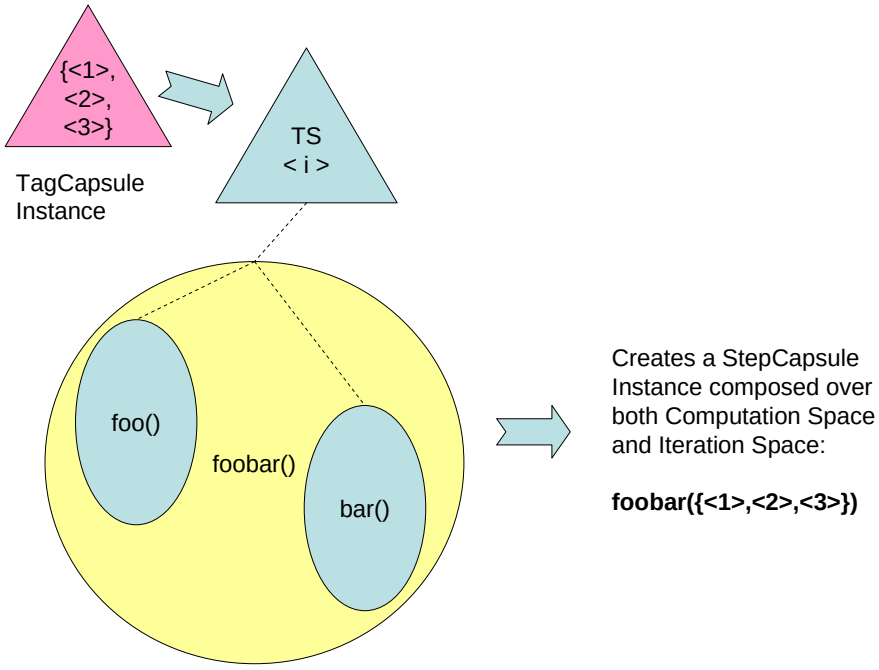


Figure 10: An example of a TagCapsule instance creating a composition over Iteration Space on a coarse-grain StepCapsule Space composed over Computation Space.

5.2 Computation-Major Serialization

A Computation-Major Serialization schedule (Figure 11) executes dual composed Step-Capsule Spaces by keeping locality between the composition over computation space or different StepCapsule Spaces as opposed to giving preference to locality between the composition over iteration space.

Computation-Major serialization could be especially useful when the total code size of all the computations in the composed coarse-grain StepCapsule Space is small enough to fit in the processor code cache. Such low code footprint could potentially give choosing computation-major serialization better performance than iteration-major serialization if the data footprint of an iteration space composed data instance is large and cannot fit in the processor data cache.

```
foobar({<1>,<2>,<3>}) {
  foo(<1>);
  foo(<2>);
  foo(<3>);

  bar(<1>);
  bar(<2>);
  bar(<3>);
}
```

```
foobar({<1>,<2>,<3>}) {
  foo(<1>);
  bar(<1>);

  foo(<2>);
  bar(<2>);

  foo(<3>);
  bar(<3>);
}
```

Figure 11: *(left)*: Iteration-Major Serialization Mode, *(right)* Computation-Major Computation-Major Serialization Mode

5.2.1 Limitations of Computation-Major Serialization

The following property must hold for dual composed StepCapsule instances that require execution in computation-major mode: Dimensional slicing must be disabled for all *Partially Specified Input Edges* (PSIE). The reason for this restriction is because slicing creates a dependency between iteration instances as all operations on data (such as *get()*) must occur at the same granularity for the same collection of instances (granularity preservation property; Section 4.6). Therefore if slicing is enabled for PSIEs, they must have access to all iteration instances involved in a slicing operation. However, in Computation-Major mode, iteration space compositions are decomposed into their finest Tag instances before execution. The decomposition of iteration space compositions essentially *hides* iteration instance from each other. For a sliced PSIE, this hidden iteration space information prevents the slicing operation to be performed.

5.3 Iteration-Major Serialization

Iteration-Major Serialization schedule (Figure 11) executes dual composed StepCapsule instances by keeping locality between the composition over iteration space. There are no limitations to task-graph formations when executing in Iteration-Major mode.

An Iteration-Major serialization schedule could give better performance when the total

data size of an iteration-space composed coarse-grain data instance is able to fit in the processor data cache. The appropriate data footprint of coarse-grain data would give Iteration-Major better performance than Computation-Major serialization if the code footprint of the composed computation cannot fit in the processor code cache.

5.4 Choosing between Serialization Schedules

The choice between Iteration-Major and Computation-Major schedules is made dynamically at runtime when a dual composed StepCapsule instance is about to be executed. In the current implementation, we ask the user to provide a serialization choice for each coarse-grain StepCapsule Space when specifying the StepCapsule Space hierarchy, which allows for deciding between schedules only at startup time. The current implementation can be changed by providing a function instead that is evaluated for each instance of the computation space composed StepCapsule instance to select the serialization schedule. More details on serialization schedules is provided in a later chapter (see Section 8.7.3).

CHAPTER VI

BENEFITS OF COMPOSABILITY

6.1 General Benefits of Composability

As described in Section 3.2, composability is useful in reducing complexity in serial programming languages (functional and procedural). However, composability plays an extended role when it is applied to a parallel programming model.

1) Composability enables the *expression of locality*. Whenever a possible StepCapsule Space can be constructed, it creates a possibility to express locality between all Capsule objects within that parent StepCapsule instance. Such locality awareness is especially beneficial in parallel programming models as it can create opportunities for optimization. Locality is achieved by using the composed StepCapsule instance as the unit for distribution.

2) Composability enables *automatic constrained garbage collection* (GC). Automatic GC is an interesting mechanism that is inherent in the procedural style used for composing coarse-grain computations. Constrained GC refers to GC due to *scoping* (visibility) rules of a composed computation. Since all ItemCapsule Spaces within a parent StepCapsule Space are only visible to StepCapsule spaces also within the same parent, the *intermediate* ItemCapsule Instances can be GC'ed when the parent StepCapsule instance has been completely executed (*i.e.,* the StepCapsule instance has reached the *executed* state).

3) Composability allows *check-pointing* at the granularity level of a composed computation. Even though we have not implemented check-pointing in the current Capsules runtime, it is a natural extension of features already present in the runtime. Check-pointing has benefits such as recovery and migration, and can especially useful in debugging. With the notion of objects of varying granularity, check-pointing can also be performed at a varying granularity to control the fidelity of information captured by the runtime.

4) Composability allows *Debugging* to be done at the granularity level of a composed computation. Debugging again, is a natural extension of the check-pointing feature described above.

5) Composability allows *reducing* system overhead by *increasing* the serial execution granularity of computations. The consolidated system overhead incurred by coarse-grain executions is always smaller than if the finer-grain constituent StepCapsule instances executed in parallel. The downside of composing computations for coarser-grain serial execution is the loss of potential finer-grain parallelism. However, this disadvantage is an acceptable trade-off for higher performance in the presence of limited hardware concurrency (Section 2.5).

6.2 Benefits of Composing over Computation Space

The benefits of composition over computation space stem largely from its hierarchical structure. Having a hierarchical composition mechanism over computation space to create coarse-grain computations has the following benefits:

(1) Enables a simple *hierarchical decision making process*.

A hierarchical decision making process can be beneficial especially in a distributed runtime. Decisions made at a higher hierarchy level (coarser-grain) only affect objects lower in the hierarchy (finer-grain). Such a hierarchical decision making process may be sub-optimal but it is more efficient that a global decision process that would otherwise require taking into account all possible capsules.

(2) Enables decisions to be made on one, multiple, or all levels of the hierarchy.

Combining benefits (1) and (2) provides the following possibilities for making decisions at the various computation space compositions, also known as levels, of the StepCapsule Space hierarchy:

(a) Execution can occur at any level of the StepCapsule Space hierarchy.

The runtime (or user) may choose not to execute serially at the coarse-grain level of a capsule but instead execute at the fine-grain level of its inner capsules. The execution level or execution granularity in a StepCapsule hierarchy is simply defined by activating execution at any level for a given StepCapsule instance. The execution of the selected StepCapsule occurs *serially* by the dynamic serialization schedule for that StepCapsule instance.

(b) Hierarchical distribution scheme can preserve locality between a parent StepCapsule instance and inner Capsule instances in a StepCapsule Space hierarchy.

A hierarchical distribution scheme in essence allows a coarse-grain StepCapsule instance to acquire a sub-set of resources that are used by its inner StepCapsule spaces. To visualize this hierarchical distribution technique, assume the entire program to be *mapped* to all available resources. Then as coarse-grain StepCapsules are instantiated, they are distributed to a sub-set of the available resources. All fine-grain capsule instances contained inside this coarse-grain capsule can be mapped to any of the resources assigned to the coarse-grain StepCapsule instance. The hierarchical resource distribution policy is not currently implemented but is a possible feature for a distributed-memory implementation of Capsules runtime.

A hierarchical distribution mechanism would call a distribution function at *all* but the *levels below* the execution-level in the hierarchy. From the execution level of the hierarchy to the finest-grain Capsule Spaces, no more distribution events need to be generated as the computations below the execution level are serially executed on one location. A distribution function would return multiple resources for a given StepCapsule Instance, and this resource set would be made available to that StepCapsule instance and all its inner Capsule instances. Distribution at the inner level of the StepCapsule hierarchy would determine where the actual instances would execute. At the execution level, the distribution function

would always returns a single resource element from the set of available resources available to its parent.

(c) Automatic Constrained GC can occur at all levels where ItemCapsule Spaces exist in the StepCapsule Space hierarchy.

Automatic Constrained GC is a unique feature that is inherent in the hierarchical structure of computation composition. It provides an aggressive GC mechanism that clears data as soon as its immediate consumers are done using it. Moreover, the automatic GC mechanism can be combined with a *Reference Count* based GC mechanism [47] by providing an *item2referencecount()* function to the runtime to enable more aggressive GC. It is important to note that such an optimization is only possible when consumers of an item are known *a priori* at item production.

(d) Check-pointing can occur at any or all levels of the StepCapsule hierarchy.

Check-pointing in a hierarchical computational structure enables events to be captured at varying granularity. For example, check-pointing can either be done at a higher fidelity for fine-grain computations, or at a lower fidelity at the coarse-grain boundary of composed StepCapsule instances. Such ability to vary the granularity of the check-pointing mechanism could potentially save on large overheads that would be incurred for a mechanism with frequent check-pointing events.

(e) Debugging can occur again at any level of the StepCapsule hierarchy.

Debugging at different levels of the hierarchy is again a natural extension of the check-pointing ability that can be performed for debugging also at various granularities.

6.3 Benefits of Composing over Iteration Space

(1) Increased locality between Capsule instances.

When computations are composed over iteration space, they also consume data that is composed over iteration space. In other words, when coarse-grain computations consume data, a coarse-grain data instance that contains multiple data instances can be gotton together to satisfy the data dependency requirements. Such coarse-grain data instances improve data-locality for coarse-grain computations. The same is true when producing data from iteration space composed computations. Multiple computation instances emit multiple data items, which can be composed and emitted together, thereby increasing the data-locality for the next consuming computation.

(2) Decreased cost for Garbage Collection (GC).

Since GC can be done at the coarse-grain ItemCapsule level instead of the finest-grain Item instance level, it reduces the total number of GC operations required during program execution. Fewer GC operations again helps reduced overall system overhead cost.

(3) Decreased runtime overhead cost.

As discussed in Section 2.6, increased execution granularity implies decreased runtime overhead at the cost of reduced parallelism. In particular, iteration space compositions lead to multiple reductions in overheads (see Section 4.4). For example, iteration space compositions reduce the total number of parallel computation instances in the runtime. A reduction in parallel computations in turn leads to a reduction in book-keeping costs, the total distribution and scheduling cost and also helps reduce the total number of synchronization points required during the execution of the entire program. All these factors help reduce the total parallelization overhead incurred during the execution of a parallel program.

CHAPTER VII

CAPSULES APPLICATION PROGRAMMING INTERFACE (API)

7.1 Introduction

In this chapter, we describe the C++ Application Programming Interface (API) for the Capsules parallel programming model. The API can be logically divided into two major parts. The first half of the API deals with constructing an application task-graph, with a user-defined composition hierarchy over computation space. The second half of the API deals with writing finest-grain StepCapsule Space stepper code, that includes *put()/get()* calls to communicate data and produce TagCapsule instances that initiate further parallel computations. The fine-grain Stepper code is also where the dynamic composition over iteration space occurs.

7.2 API: Constructing task-graph with computation hierarchy

The application task-graph is described using the three basic software primitives, namely: The *StepCapsule Space*, the *ItemCapsule Space* and the *TagCapsule Space*. The entire program is first defined by instantiating a *default* StepCapsule Space that represents the entire application computation (Figure 12).

```
StepCapsuleSpace_t* myApp = new StepCapsuleSpace_t("AppName");
```

Figure 12: API: Creating the outer most default StepCapsule Space.

All other space objects are added into this *default* StepCapsule Space to construct the application task-graph. As a result, the *default* StepCapsule space is the root of the application StepCapsule hierarchy. The APIs used to add a space object to a StepCapsule Space are shown in Figure 13.

```
1  class StepCapsuleSpace_t {
     StepCapsuleSpace_t* addStepCapsuleSpace (
3              const string& newStepCapsuleSpaceName,
               const string& prescriberTagCapsuleSpaceName,
5              StepCapsuleFunction_t func = 0,
               s2r_t s2r = 0);
7    void addItemCapsuleSpace (
               const string& newItemCapsuleSpaceName,
9              const string& prescriberTagCapsuleSpaceName);
     void addTagCapsuleSpace(
11             const string& newTagCapsuleSpaceName,
               const string [] axisName);
13 };
```

Figure 13: API: Adding Step/Item/Tag Capsule Spaces to construct a StepCapsule Space hierarchy.

7.2.1 Adding child StepCapsule Spaces

The *addStepCapsuleSpace()* API call allows the hierarchical composition over computation space. Note that this API requires a top-down construction of the StepCapsule Space hierarchy, *i.e.*, the coarse-grain computations need to be defined first, followed by inserting the finer-grain computations within them.

The first parameter, *newStepCapsuleSpaceName*, uniquely identifies a StepCapsule Space. The second parameter, *prescriberTagCapsuleSpaceName*, is the name of the Tag-Capsule Space that prescribes the new StepCapsule Space. If an inner StepCapsule Space needs to be parametrized by the TagCapsule Space of the parent StepCapsule Space, a second API is available that omits this prescriber's name (not shown here for brevity). In such a case, the inner StepCapsule Space is said to be *parent-prescribed*. The third parameter, *func*, is used only if the StepCapsule Space represents the finest-grain indivisible computation. The *func* parameter is a function pointer to the user-defined finest-grain Stepper code described in the next section. The last parameter, *s2r*, represents an optional function pointer to a resource distribution policy. It is used to make decisions on what PE StepCapsule instances are executed on. The runtime also provides a default *round-robin* distribution scheme if no distribution policies are defined by the user.

7.2.2 Adding child ItemCapsule Spaces

The *addItemCapsuleSpace()* API call helps add ItemCapsule Spaces to the application task-graph. An ItemCapsule Space follows scoping rules and is visible to StepCapsule Spaces that are either its siblings or are children of its siblings.

The parameters to this call have the exact same semantics as those for adding StepCapsule Spaces (Section 7.2.1).

7.2.3 Adding child TagCapsule Spaces

Inner TagCapsule Spaces are added to application task-graph to parametrize other inner Step/Item Capsule Spaces. These TagCapsule spaces also represent new iteration dimensions for either computation or data.

Again, the first parameter, *newTagCapsuleSpaceName* represents the space name. The second parameter, *axisName* represents the list of dimensions spanning the TagCapsule Space. The dimension at array index $i - 1$ corresponds to the Tag instance dimension i, also represented at depth i in a TagCapsule Instance tree (Section 2.3).

7.2.4 Specifying Producer/Consumer Relationships

In this section, we describe API calls (Figure 14) used to create edges in the application task-graph that represent producer/consumer relationships between the finest-grain StepCapsule Spaces and other Item/Tag Capsule Spaces. Since edges are always expressed with respect to the producing/consuming computation, the API calls for creating edges are also encapsulated in the *StepCapsuleSpace_t* class.

All API calls simply require the name of the consumed or produced Item/Tag Capsule Space. Except for the *addConsumerItemCapsuleSpace()* API call that has an optional *grainDefs* parameter used in dimensional reduction (PSIE), all other API calls are the same.

```
1  class StepCapsuleSpace_t {
     int addConsumerItemCapsuleSpace(
3            const string& itemCapsuleSpaceName,
             vector<GrainDefFunction_t>& grainDefs);
5    int addProducerItemCapsuleSpace(
             const string& itemCapsuleSpaceName);
7    int addProducerTagCapsuleSpace(
             const string& tagCapsuleSpaceName);
9  };
```

Figure 14: API: Creating Producer/Consumer Relationships between StepCapsule Spaces and Item/Tag Capsule Spaces.

7.2.4.1 Dimensional Reduction and Dimension Definition Functions

The optional second *grainDefs* parameter in *addConsumerItemCapsuleSpace()* function represents a collection of *Dimension Definition Functions* (DDF). These are used only when a dimensional reduction (Section 4.5; Rule (6)) occurs between the consuming StepCapsule Space and the target ItemCapsule Space. Dimensional reduction occurs when an ItemCapsule Space has U (where $U \geq 1$) unmatched missing dimensions with respect to the dimensions of its consuming StepCapsule Space. These unmatched ItemCapsule Space dimensions are defined by the *grainDefs* functions as they cannot be defined by the parametrizing TagCapsule instance of the consuming StepCapsule instance. Each dimension definition function can define D_i dimensions. The total dimensions defined by all DDFs should equal the number of missing U ItemCapsule dimensions, *i.e.,* $U = \sum D_i$.

For example, assume a StepCapsule Space iterates over dimensions $< i, j >$. Now assume that this StepCapsule Space consumes from an ItemCapsule Space iterating over dimensions $< j, k >$. Here, for any given value of j, corresponding values for k need to be defined so that the StepCapsule can consume its data. Such dimension definition is done with the help of these DDFs. The function prototype for a DDF is shown in Figure 15.

The first parameter *env* allows the DDF access to data over Fully Specified Input Edges

```
1 typedef void (grainDefFunc)(Environment_t&          env,
                              const Tag_t&            tag,
3                             vector<TagCapsule_t>&   grainTC);
  typedef grainDefFunc* GrainDefFunction_t;
```

Figure 15: API: Function prototype for a Dimension Definition Function.

(FSIE). The DDF can use this data to define the missing dimensions. The *tag* parameter contains dimensions that have already been defined (*i.e.,* the fully specified dimensions). The last argument *grainTC* is used to store the output, the defined dimension values grouped into granularity sets. Note that multiple dimensions can be defined by one DDF. The user must maintain consistency throughout the application in emitting together dimensions that are also defined together.

7.3 API: Finest-grain StepCapsule Space and Composition over Iteration Space

The function prototype for the user-defined finest-grain stepper function is shown in Figure 16. The first parameter, *env*, is a handle to the environment for the given Step instance. It provides access to relevant ItemCapsule and TagCapsule Spaces that are within scope. The second parameter, *tag*, represents the unique identifier Tag instance for the Step instance.

```
void StepCapsuleFunction_t(Environment_t& env, const Tag_t& tag);
```

Figure 16: API: Function prototype for the user-defined stepper function.

The *Environment_t* class provides the API in Figure 17 to allow the programmer to interact with other ItemCapsule and TagCapsule Spaces from within the finest-grain stepper function.

It is important to note that the runtime retrieves the required data (ItemCapsule instances) before the finest-grain stepper function is ever executed. Performing such *gets()* in advance is possible because the runtime is aware of the input data-dependencies of each

```
1  class Environment_t {
     const Item_t* getItem(
3              int inConnID);
     const ItemCapsule_t* getItemCapsule(
5              int inConnID);
     void putItem(
7              int outConnID,
               const Item_t* item);
9    void putItemCapsule(
               int outConnID,
11             const ItemCapsule_t* ic,
               const TagCapsule_t& tc);
13   void putTag(
               int outConnID);
15   void putTagCapsule(
               int outConnID,
17             const TagCapsule_t& tc);
   };
```

Figure 17: API: Data access calls used by finest-grain stepper functions to *put()* and *get()* Item and ItemCapsule instances over FSIEs/FSOEs and PSIEs/PSOEs respectively.

StepCapsule instance via the producer edge information and the implied *one-to-one* semantics between the dimension values of the StepCapsule Spaces and its input ItemCapsule Spaces. Therefore, the *get()* calls provided in the Environment class simply retrieve the specific item from the environment's pre-fetched list of data for the specified edge.

The first two API calls *getItem()* and *getItemCapsule()* are used to retrieve data into the Step instance. The *getItem()* call is used for *fully specified input edges* (FSIEs), where all dimensions of the ItemCapsule Space match with the dimensions of the StepCapsule Space. The *getItemCapsule()* call is used for *partially specified input edges* (PSIEs) that denote a dimensional reduction. As dimensional reduction edges always result in multiple Items, an ItemCapsule consisting of all Items is returned.

The *putItem()* and *putItemCapsule()* API calls emit data out from the fine-grain Step instance into the parallel application environment. Similar to the *getItem()* call, the *putItem()* call is used for *fully-specified output edges* (FSOEs), where all dimensions of the ItemCapsules match with the dimensions of the StepCapsule Space and *without* any extra dimensions in the StepCapsule Space. (Section 4.5, Rule (1-2)). The *putItemCapsule()* call

is used for *partially-specified output edges* (PSOE) where dimensional expansion occurs. Dimensional expansion represents StepCapsule Spaces creating one or more new Tag dimension that do not exist in the StepCapsule Space's own prescribing TagCapsule Space.

The *putTag*()* calls have the exact same semantics as the *putItem*()* calls.

7.4 Dimensional Expansion and Composition over Iteration Space

Edges that denote dimensional expansion also must specify the granularity of the new dimensions. StepCapsule spaces with dimensional expansion are therefore required to create the coarse-grain capsules by composing over the new iteration space dimensions. These composed coarse-grain Item/Tag Capsule instances are emitted via the *putItemCapsule()* and *putTagCapsule()* API calls.

7.5 Side-effect free property of Composed Computations

One property of composed computations is that they are side-effect free, except for the data-access specified in the task-graph. Such a property is derived from the fine-grain function calls that the composition is made from that also have the same side-effect free property. The property is retained even when further compositions are done to create more coarse-grain computations.

The side-effect free property allows an executing coarse-grain StepCapsule instance to be terminated and re-executed without fear of unspecified behavior. Such property allows a check-point-restart model useful in many ways. In enables a simple execution model where any execution can be terminated without maintaining state due to, for example, unavailable data. The execution can be re-initiated when the data does become available. The property also enables simple and efficient task migration, where no state information need be transported.

7.5.1 StepCapsule Spaces *with* Side-effects

It is important to note that there can be StepCapsule Spaces that are explicitly designed to have side-effects, *e.g.,* a *primer* StepCapsule Space that writes to disk. In such a case, the side-effect free property is disabled, and can no longer be propagated to coarse-grain StepCapsules that contain such fine-grain StepCapsule Spaces.

CHAPTER VIII

CAPSULES RUNTIME IMPLEMENTATION

8.1 Introduction

In this chapter we describe in depth the internals of the Capsules runtime implementation. We begin by describing a high-level description of the execution model, followed by details of the automatic GC mechanisms, program termination protocol, and the debug runtime library. We then delve into the internal runtime data-structures that support the programming model. We end this chapter with a discussion of the serialization schedules that execute coarse-grain computations serially.

8.2 Execution Model for a SMP/CMP machine

The current C++ Capsules runtime is implemented for a SMP/CMP machine. It is based on a simple execution model of work queues, also known as run-queues. Each processor, or processing core, is considered as a separate Processing Element (PE), with each PE assigned a dedicated work queue and a work thread to execute tasks in parallel. Each work thread continuously pops StepCapsule instances from the work queue head to execute them, whereas new StepCapsule instances are constantly being inserted at the work queue tail.

Each StepCapsule instance is either composed of only one finest-grain StepCapsule instance or is a computation space composed StepCapsule instance consisting of multiple Step/Item/Tag Capsule instances within it. If it is an indivisible finest-grain StepCapsule instance, the execution invokes the serialization schedule of the StepCapsule and executes all inner finer-grain StepCapsule instances parametrized by the TagCapsule instance. However, if the StepCapsule instance is a composed StepCapsule instance, it can be operated on in one of two ways. It can either be serialized, that is all internal StepCapsules be executed

serially with respect to each other, or it can act as a GC container. If the StepCapsule is a GC container, the inner StepCapsule instances in this case are executed in parallel in work queues similar to non-composed StepCapsule instances.

A GC thread keeps track of computation space composed StepCapsule instances that are not being serialized and used only as GC containers. The GC thread checks for the GC condition for such StepCapsules at pre-defined intervals. After program termination, the GC condition is checked for any remaining StepCapsule GC containers.

8.3 Automatic Garbage Collection

The GC container StepCapsule instance keeps track of executed inner StepCapsule instances. Once all inner StepCapsule instances are done executing, the GC container and all Step/Item/Tag Capsule instances contained within it are garbage collected. The StepCapsule instance and the StepCapsule hierarchy (Section 3.3.1) therefore enable automatic garbage collection within the Capsules runtime.

Every application execution is encapsulated in a default StepCapsule GC instance that contains all application StepCapsule instances.

8.3.1 Other GC mechanisms

A more aggressive GC mechanism that uses reference counting can also be employed to reduce the memory footprint of the application data stored by the runtime. In a previous study of garbage collection algorithms in the Space-Time programming model called *Stampede* [47, 25, 58], a Reference Counting (REF) algorithm was used to aggressively GC time-indexed data when all consumers of an item signaled being done with the item. However, it is important to note that reference counting based GC mechanisms only work when the number of consumers is known at the time of data production. They do not work when the number of consumers is unknown at such time and therefore a consumer count cannot be determined. In such cases, the conservative automatic GC mechanism of the StepCapsule instance GC container can act as a fall back mechanism to clean up used data.

In other words, a reference counting based GC mechanism can act as an optimization in certain scenarios and can be used along side the conservative automatic GC mechanism.

8.4 Program Termination

Program termination in the Capsules runtime is achieved when all work threads participating in the parallel execution are done executing StepCapsule instances and go back to the *sleep state*. PE_0 is fixed as the thread that determines this termination state. The termination algorithm is described as follows: A counter representing the number of sleeping threads $num\_sleeping$ is maintained by PE_0. Termination is therefore achieved when this counter reaches the number of threads participating in the parallel execution *i.e.,*

$$Terminate : (num\_sleeping == num\_PEs)$$

Whenever a thread is woken from its sleep state, the $num\_sleeping$ counter is decremented. Note that work threads are only woken when a StepCapsule instance is inserted into an empty work queue. Work threads remain in the woken state until the work queue becomes empty again, *i.e.,* there are no more tasks to execute. When a thread goes back to sleep, the $num\_sleeping$ counter is incremented.

8.5 Debug runtime Library

The Capsules runtime library can be compiled with debugging features that would otherwise add to the runtime overhead. Debugging features include enabling prescription of ItemCapsules, which is disabled by default as data prescription only acts as a verification mechanism to check correctness of data consumption. Other debugging features include assertion checks for well formed Tag/Item Capsule instance structures passed in as arguments to the Capsules API. Logging of special execution events can also be enabled to get information about work-load imbalance and total execution time (Section 9.1.1) for a given application.

These debugging features are switched off, except for logging, to capture the experiment data reported in Chapter 9.

8.6 Internal Data-structures

In this section we describe the internal data-structures representing all objects in the Capsules runtime. First, the data-structures used to encapsulate all the three Spaces are described. Next, the Pool data-structures are described, which hold the individual Capsule instances. Next, the actual Capsule Instance object data-structures are explained. Lastly, the Environment data-structure is described, that lends access to unpacked input and output data to the fine-grain user-defined stepper code.

8.6.1 TagCapsuleSpace_t

The *TagCapsuleSpace_t* class (Figure 18) contains information about the iteration space dimensions that it represents and the Step/Item Capsule Spaces that it parametrizes.

```
class TagCapsuleSpace_t {
private:
  const string                        name;
  vector<string>                      axes;
  map<string, StepCapsuleSpace_t*>    prescribedStepCapsuleSpaces;
  map<string, ItemCapsuleSpace_t*>    prescribedItemCapsuleSpaces;
protected:
  static const TagCapsuleSpace_t NullTagCapsuleSpace;
};
```

Figure 18: TagCapsuleSpace_t data-structure

Each TagCapsuleSpace is uniquely identified by a *name* string. The dimension names are stored in *axes* vector. The *prescribedStepCapsuleSpaces* and *prescribedItemCapsuleSpace* list all the spaces prescribed by this TagCapsule Space. The *NullTagCapsuleSpace* is a static Space that parametrizes the outer-most *default* StepCapsule Space and all its parent-prescribed inner spaces.

```
1  class ItemCapsuleSpace_t {
   private:
3    const string                name;
     const StepCapsuleSpace_t* parent;
5    const TagCapsuleSpace_t*  prescriber;
   };
```

Figure 19: ItemCapsuleSpace_t data-structure

8.6.2 ItemCapsuleSpace_t

The *ItemCapsuleSpace_t* class (Figure 19) is a simple data-structure that includes, again, the unique *name* for the space and a pointer, *parent*, to the StepCapsule Space that contains it. It also encapsulates a pointer, *prescriber*, to its prescribing TagCapsule Space.

8.6.3 StepCapsuleSpace_t

The *StepCapsuleSpace_t* class (Figure 20) is one the most important data-structures used by the Capsules runtime. It encapsulate all information required to enable parallel execution of a Capsules program. The *StepCapsuleSpace_t* data-structure can be sub-divided into two parts. The first part is populated during the application task-graph construction, whereas the second part is populated after automatic analysis of the task-graph. The analysis data is used during the efficient serial execution of a coarse-grain StepCapsule instance.

Similar to all other Spaces, *StepCapsuleSpace_t* also contains a unique identifier string variable, *name*. The *prescriber* points to the TagCapsule Space prescribing this space. The *prescriber* can be set to *null* to represent a parent-prescribed relationship. Next, the *func* function pointer is used if the StepCapsule Space represents actual user-defined code steppers. If the StepCapsule Space is composed over computation space consisting of other Spaces, this function pointer is set to *null*. The *s2r* variable is optionally used to specify a user-defined StepCapsule instance distribution policy. Distribution policies specify the resource id where a StepCapsule instance is to be executed. A default distribution policy can be used if no user-defined policy is specified.

```
class StepCapsuleSpace_t {
private:
  const string                        name;
  const StepCapsuleSpace_t*           parent;
  const TagCapsuleSpace_t*            prescriber;
  const StepCapsuleFunction_t         func;
  const s2r_t                         s2r;
  const ccf_t                         ccf;
  bool                                serializationMajor;
  map<string, StepCapsuleSpace_t*>    stepCapsuleSpaces;
  map<string, TagCapsuleSpace_t*>     tagCapsuleSpaces;
  map<string, ItemCapsuleSpace_t*>    itemCapsuleSpaces;

  // unanalysed consumer edges
  vector<ItemConsumerEdge_t>          consumerICS;
  // analyzed producer edge information
  vector<OutItemEdge_t>               FSE_OutItem;
  vector<OutItemEdge_t>               PSE_OutItem;
  vector<OutTagEdge_t>                FSE_OutTag;
  vector<OutTagEdge_t>                PSE_OutTag;
  // analyzed consumer edge information
  InItemEdgeArray2D_t                 firstFSE_InItem;
  InItemEdgeArray2D_t                 firstPSE_InItem;
  InItemEdgeArray2D_t                 lastFSE_InItem;
  InItemEdgeArray2D_t                 lastALL_InItem;
  InItemEdgeArray2D_t                 firstANDinterANDlastFSE_InItem;
  InItemEdgeArray2D_t                 firstANDinterANDlastALL_InItem;
  InItemEdgeArray3D_t                 lastFSEoverlapWithPSEInfo;
  vector<StepCapsuleSpace_t*>         schedule;
};
```

Figure 20: StepCapsuleSpace_t data-structure

The next five variables namely, *ccf*, *serializationMajor*, *stepCapsuleSpace*, *tagCapsuleSpaces* and *itemCapsuleSpaces* are only used if the StepCapsule Space is composed over computation space.

The *ccf* function pointer represents the policy used for Composition over Computation Space. The user-defined *ccf* function determines when a computation space composition is to be serialized. Once a coarse-grain StepCapsule is serialized, all internal StepCapsules are also inherently serialized. The *serializationMajor* variable determines if the StepCapsule is serially executed in Iteration-Major or Computation-Major mode. More details about serial execution modes for coarse-grain StepCapsules composed over computation space is given in Section 8.7.3.

Next, the three variables, namely *stepCapsuleSpaces*, *tagCapsuleSpaces* and *itemCapsuleSpaces*, store pointers to inner spaces that together make the composed StepCapsule Space. These are used during the analysis phase and also during runtime execution to determine what inner computations and data elements make up the StepCapsule Space and what their relationships are with each other and their composed parent StepCapsule Space.

The next variable *consumerICS* stores input edges from ItemCapsule Spaces. These edges are later analyzed to determine dimensional relationships between the consumed ItemCapsule Spaces and the consuming StepCapsule Space in order to find boundary points where coarse-grain ItemCapsules can be retrieved to minimize the total number of synchronization points required during parallel execution (Section 4.4).

The next four variables namely, *FSE_OutItem*, *PSE_OutItem*, *FSE_OutTag* and *PSE_OutTag* represent the producer output edges from the StepCapsule Space. They are categorized automatically as Partially Specified and Fully Specified when being added by the consumer. If the StepCapsule Space is composed over computation space, the input edge classification is inferred from the relationships of the composed StepCapsule Space and the consumed ItemCapsule Space. Note that the user does not have to explicitly create edges between the computation space composed StepCapsule Space and the produced/consumed spaces. These are automatically created by determining the intermediate compositions between the finest-grain StepCapsule Spaces and the consumed/produced spaces.

As described in Section 4.4.1, input edges are analyzed and separated into S sets (where S is the dimensionality of the StepCapsule Space) to derive the Dimensional Boundary information. Each set can be categorized as either: (1) *First Dimension Dependency*, (2) *Intermediate Dimension Dependency*, (3) *Last Dimension Dependency*.

The variables *firstFSE_InItem* and *firstPSE_InItem* contain the *First Dimension Dependency* edges. The variables *lastFSE_InItem* and *lastALL_InItem* contain *Last Dimension Dependency* edges. The variables *firstANDinterANDlastFSE_InItem* and *firstANDinterANDlastALL_InItem* contain edges that fall in all three categories.

The *lastFSEoverlapWithPSEInfo* variable is used to store the *Last Dimension Dependency* FSIEs that have all matching dimensions with a given PSIE. The dimension dependency information is used to retrieve data over FSIEs to specify missing data-dependent dimensions of a PSIE. These data-dependent dimensions are specified using dimension definition functions (Section 7.2.4.1).

Lastly, the *schedule* variable consists of a non-blocking schedule for executing inner StepCapsule Spaces. The non-blocking schedule is constructed by analyzing the data-dependencies between internal computations and resolving any conflicts. The schedule is used to serially execute the coarse-grain computation space composed StepCapsule instance.

8.6.3.1 Edge Information

The *OutTagEdge_t*, *InItemEdge_t* and *OutItemEdge_t* data-structures (Appendix A) contain information about input and output edges from a StepCapsule Space. The edge data-structure contains a pointer to the input/output Item/Tag Capsule Spaces. It also contains a unique integer index that allows access to data storage arrays for that edge in the *Environment_t* object. The data-structures also inform if edges cross the parent StepCapsule Space's boundary, in which case they are classified as *external*. An external edge index that denotes the same edge in the parent StepCapsule Space is also stored. Information about external edges and their relative indexes are computed during the application task-graph edge analysis phase. Information like the *dimension definition functions* are also stored for Partially Specified Input Edges (PSIEs).

8.6.4 ItemCapsulePool_t

The *ItemCapsulePool_t* data-structure (Figure 21) is a container used to hold ItemCapsule instances. Besides containing live ItemCapsules that have already been produced, *ItemCapsulePool_t* also has information on StepCapsule instances that are blocked on specific ItemCapsule instances that have not yet been produced.

```
class ItemCapsulePool_t {
private:
  class ItemCapsuleInfo_t {
  public:
    const TagCapsule_t      tc;
    const ItemCapsule_t*    ic;
    bool                    prescribed;
    vector<StepCapsule_t*> blockedStepCapsules;
  };

  const ItemCapsuleSpace_t*      space;
  const StepCapsule_t*           parent;
  map<Tag_t, ItemCapsuleInfo_t> pool;
  mutable pthread_mutex_t        mutex;
};
```

Figure 21: ItemCapsulePool_t data-structure

Each pool contains a variable, *space*, pointing to its ItemCapsule Space data-structure that contains static information about the space. Each pool also has a pointer to the *parent* StepCapsule instance. Access to the parent is required to access data in the StepCapsule instance hierarchy. The *pool* variable in *ItemCapsulePool_t* is a RedBlack-tree (RB-tree) used to store the ItemCapsule instances. The tree is indexed by the unique *Tag-key* that represents the first Item in the ordered list of Item instances in the ItemCapsule instance. There is also a *mutex* lock, which serializes parallel access to the shared *pool* of ItemCapsules.

8.6.4.1 Blocked StepCapsule Instances

Recall that the *pool* variable in *ItemCapsulePool_t* is an RB-tree containing *ItemCapsuleInfo_t* nodes. The *ItemCapsuleInfo_t* structure is designed to hold information on StepCapsule instances that may be blocked on a given ItemCapsule instance not yet produced at the point in time. Moreover, an ItemCapsule instance may be consumed by multiple StepCapsule instances, therefore, each consuming StepCapsule instance is recorded into the *blockedStepCapsule* vector. StepCapsule instances may also have multiple unavailable input data dependencies at a given point in time. To keep track of required unavailable ItemCapsules, the StepCapsule instance maintains a *numWaitingItemCapsules* counter (Section 8.6.8). When a previously unavailable ItemCapsule instance is produced, the

blockedStepCapsule vector is used to determine the consuming StepCapsule instances. The *numWaitingItemCapsules* counter in each consuming StepCapsule instance is then decremented. If this counter reaches 0, a StepCapsule instance reaches the *enabled* state, to signal the *availability* of all its known input data dependencies. The StepCapsule can now be re-inserted into its assigned execution queue effectively unblocking the StepCapsule instance. Once all StepCapsule instances in the *blockedStepCapsule* list are processed, the vector is cleared and the ItemCapsule instance production is complete.

8.6.5 StepCapsulePool_t

```
1  class StepCapsulePool_t {
   private:
3    const StepCapsuleSpace_t* space;
     StepCapsule_t*            parent;
5    vector<StepCapsule_t*>    pool;
     mutable pthread_mutex_t   mutex;
7    int                       num_pending;
   };
```

Figure 22: StepCapsulePool_t data-structure

The *StepCapsulePool_t* data-structure (Figure 22) is similar to *ItemCapsulePool_t* used to store StepCapsule instances. The *pool* itself is a simple vector, unlike the RB-tree used to store ItemCapsule instances, which require fast $\lg(n)$ operations for insertion and retrieval. The StepCapsule pool, on the other hand, does not need access to individual instances except for deletion. A *mutex* protects access for parallel insertions into the *pool*. The *num_pending* counter is used by the runtime to keep track of unexecuted StepCapsule instances in the pool and determine when they have all completed execution. The *num_pending* counter is therefore used by the parent GC container to determine its GC condition.

8.6.6 TagCapsule_t

The *TagCapsule_t* data-structure (Figure 23) is a tree as shown in Figure 3. Each tree node contains a dimension *value*, and a list of pointers, *children*, to child nodes representing the

next dimension values. The *TagCapsules_t* structure that encapsulates *children* is in fact an object that uses a smart pointer technique to auto garbage collect itself. The smart pointer mechanism removes the explicit need for a TagCapsule pool data-structure for garbage collection of TagCapsule instances.

Each *TagCapsule_t* node also contains an index, *path*, into a child node to the next dimension. Composing all dimension values pointed to by each successive *path* to the root node yields the Tag-key, the first Tag from the list of ordered Tags represented by the TagCapsule instance. The Tag-key uniquely identifies the entire TagCapsule instance.

```
class TagCapsule_t {
private:
  IntRange_t        value;
  TagCapsules_t     children;
  int               path; // index to the left most maximum depth node
public:
  static const TagCapsule_t NullTagCapsule;
};
```

Figure 23: TagCapsule_t data-structure

8.6.6.1 Integer Range: Optimization for dimensional value representation

As a memory optimization, the TagCapsule tree node allows many contiguous integer values to be represented as a compressed single integer range value. Integer ranges (Figure 24) can be used on multiple dimensions of the TagCapsule tree. However, there are limitations on how such integer ranges could be used (Section 8.6.6.2). Integer ranges in TagCapsules are enumerated into Tag instances by iterating through all integers between the *lower* and *upper* range values (inclusive). However, multi-dimensional integer ranges are enumerated into Tags by performing a cross-product between the range values of each dimension. A cross product operator also requires a precedence definition, which is implicitly defined as left to right with decreasing precedence. Therefore $IR_i x IR_{i+1}$ enumerates all Tag instances from a multi-dimensional integer range IR_i.

```
class IntRange_t {
public:
  int l; // lower
  int u; // upper
  bool isRange;
  bool isNull;
};
```

Figure 24: IntRange_t data-structure

8.6.6.2 Limitations to the Integer Range Optimization

Integer ranges can only be used on dimensions that are not further expanded on. For example, a computation *foo()* executing over dimensions $< i, j >$ may emit a new dimension $< k >$ as an integer range. However, no computation dependent on k (*i.e.,* executing over the dimensions $< i, j, k >$) can perform a dimensional expansion to expand the iteration space further. Such a restriction is required to avoid complexities in the cross-product semantics (Section 8.6.6.1) used to enumerate Tag instances from a TagCapsule tree.

8.6.7 ItemCapsule_t

```
class ItemCapsule_t {
private:
  IntRange_t                    value;
  vector<const ItemCapsule_t*> children;
  vector<const Item_t*>         data;
};
```

Figure 25: ItemCapsule_t data-structure

The *ItemCapsule_t* data-structure (Figure 25) mimics the *TagCapsule_t* data-structure except for the additional vector, *data* to hold items. The *data* variable is used only in the N^{th} dimension leaf nodes of the ItemCapsule tree.

8.6.8 StepCapsule_t

The *StepCapsule_t* data-structure (Figure 26) encapsulates StepCapsule instances within the Capsules runtime. It can be divided into two set of variables: The first variable set

```
class StepCapsule_t {
private:
  typedef map<Tag_t, const ItemCapsule_t*> ItemCapsules_t;

  const TagCapsule_t                  tc;
  StepCapsulePool_t*                  parentPool;
  int                                 queueid;
  map<string, StepCapsulePool_t*>     scPools;
  map<string, ItemCapsulePool_t*>     icPools;
  bool                                serialExe;
  bool                                serialExeRoot;

  int                                 numWaitingItemCapsules;
  mutable pthread_mutex_t             mutex;
  Environment_t                       env;
  DataTree_t*                         dt;
  map<string, ItemCapsules_t>         cachedInEdges;
  int                                 resumeScheduleIndex;
  bool                                executed;
};
```

Figure 26: StepCapsule_t data-structure

is initialized when an instance is created. The second variable set is populated with data during the serialized execution of the StepCapsule instance.

The first variable *tc* represents the TagCapsule instance that parametrizes the StepCapsule instance. A pointer to the pool that contains this StepCapsule instance is also provided. Access to the pool gives access to the parent StepCapsule instance, and also to the Space that this instance is a member of. Access to the parent StepCapsule instance is necessary to retrieve data from the StepCapsule hierarchy, whereas access to the space allows access to static information required during serial execution. The *queueid* represents the PE where the StepCapsule instance is to be executed. The variables *scPools* and *icPools* store pointers to inner pools that contain inner object instances. These variables are of course only used if the instance represents a computation space composed StepCapsule. The variable *serialExe* determines if this StepCapsule is to be serialized. Note that all fine-grain StepCapsule instances must have *serialExe* set to *true*. The variable *serialExeRoot* is used to determine if the current StepCapsule instance is the top most StepCapsule instance in the computation hierarchy that is being serialized. The *serialExeRoot* information is required

as only the serialization root StepCapsule instances are inserted into an execution queue. All inner StepCapsule instances that are part of a serialized parent are therefore not inserted in the execution queue. They are, by definition, managed and executed during the serial execution of the parent.

Now we describe variables that are used during StepCapsule execution. The first variable *numWaitingItemCapsules* is a counter representing the number of ItemCapsule instances a StepCapsule instance is blocked on. The counter is increment when a *get()* request fails during execution and decremented when a requested ItemCapsule instance is produced. When a previously requested ItemCapsule instance is produced, the blocked StepCapsule instances are removed from the ItemCapsule's *blockedStepCapsules* list. A *mutex* serializes access to this shared counter. Once the counter reaches 0, the StepCapsule is ready to execute again. The next variable *env* holds unpacked input and output data for access to user-defined stepper code. More information about the environment is given in Section 8.6.9. The next variable *dt* is only used if a StepCapsule has PSIEs. It is used to store all input ItemCapsules from both PSIEs and FSIEs. PSIEs require FSIEs to be pre-fetched as FSIEs may be required as input to dimension definition functions, which in-turn define missing dimensions to retrieve PSIE data. In cases where the StepCapsule Space does not have any PSIEs, *dt* is not used and all FSIE data, if any, is stored (and unpacked) directly in the *env*. The next variable *cachedInEdges* is used only for StepCapsules composed over computation space and only when the StepCapsule is executing in Iteration-Major mode. During Iteration-Major mode, data over external edges are cached for access to inner StepCapsules. The variable *resumeScheduleIndex* is used as an index into the schedule for serializing computations composed over computation space. Since data over PSIEs into finest-grain StepCapsule steppers is gotten lazily, PSIEs data *get()* operations may fail due to unavailability of data. Therefore, serial execution of a coarse-grain computation composed over computation space may fail, and may require the runtime to suspend execution and resume after data is available. The *resumeScheduleIndex* allows

computation space schedules to pause and resume when PSIE data is unavailable.

8.6.9 Environment_t

The *Environment_t* data-structure (Figure 27) is used to keep track of unpacked data during execution of a coarse-grain StepCapsule instance. The serialization schedule uses the Environment to hold unpacked data from coarse-grain input ItemCapsule instances and provides access to them from the user-defined stepper functions. The Environment also holds emitted Items and Tags before they are packed into coarse-grain Item/Tag Capsule instances.

```
class Environment_t {
private:
  StepCapsule_t*        parent;
  // FSIE (Item Edges)
  const Item_t*         initemdata        [MAX_IN_EDGES];
  //  PSIE (Item Edges)
  ItemCapsuleArray1D_t  initemcapdata     [MAX_IN_EDGES];
  // PSOE (Tag Edges)
  TagCapsuleArray2D_t   outtagcap         [MAX_OUT_EDGES];
  int                   numTagSlices      [MAX_OUT_EDGES];
  // FSOE (Item Edges)
  ItemArray1D_t         outitemdata       [MAX_OUT_EDGES];
  // PSOE (Item Edges)
  ItemCapsuleArray2D_t  outitemcapdata    [MAX_OUT_EDGES];
  TagCapsuleArray2D_t   outitemcaptagcap  [MAX_OUT_EDGES];
  int                   numItemSlices     [MAX_OUT_EDGES];
  // Leaf-Node Counter
  int                   leafNodeIndex;
};
```

Figure 27: Environment_t data-structure

Now we describe the individual variables: The *parent* pointer enables access to the *StepCapsule_t* instance that contains this *Environment_t* data-structure. The *Environment_t* allows access to the data in the StepCapsule instance hierarchy. The *initemdata* holds data over the *Fully Specified Input Edges* (FSIE). Likewise, the *initemcapdata* holds data over *Partially Specified Input Edges* (PSIEs).

Similarly, the *outtagcap* holds TagCapsules emitted over *Partially Specified Output Edges* (PSOE). The maximum number of TagCapsules emitted by any stepper call in the

coarse-grain StepCapsule instance is stored in *numTagSlices*. As the name suggests, this number is used to signify the number of slices generated if the *slicing* operator (see Section 4.8) is enabled while performing a dimensional expansion on the PSOE. The *outitemdata* holds data emitted over *Fully Specified Output Edges* (FSOE). The *outitemcapdata* holds data emitted over PSOEs, with its identifying Tags being stored in *outitemcaptagcap*. The *numItemSlices* is again used to keep a count on the maximum number of ItemCapsules emitted by any of the number of stepper function calls made during the execution of the StepCapsule instance. The variable is again used if slicing is enabled for the edge. Lastly, the *leafNodeIndex* keeps track of TagCapsule leaf node id being traversed during the execution of the StepCapsules composed over iteration space. The *leafNodeIndex* variable is used to access the dynamically allocated data holders for the PSOEs.

8.6.9.1 Alternate Environment_t location

An alternate location for the Environment variable would be outside the *StepCapsule_t* and in the work threads since the Environment is only used when executing a StepCapsule instance. The advantage of an *Environment_t* stack per work thread is a lower memory footprint. However, an *Environment_t* stack would be required to represent the coarse-grain StepCapsule instance hierarchy. The Environment at the top of the stack would represent the leaf StepCapsule in the StepCapsule hierarchy, whereas the Environment at the bottom would represent the coarsest-grain StepCapsule instance where serialization was initiated.

However, this optimization would not be possible if the programming model needed to support parallel execution within serialized StepCapsules composed over computation space. For example, during serialized execution of a StepCapsule sub-hierarchy, there are new resources available and the finer-grain StepCapsules (lower in the StepCapsule hierarchy) can revert to parallel execution. To allow parallel execution *within* a serialized coarse-grain execution, the execution state needs to be saved before the executing work thread can context switch. As the execution state is stored in the Environment data-structure, it must

be saved with the switched out serialized coarse-grain StepCapsule.

8.7 Serialization Schedule

In order to serially execute a coarse-grain StepCapsule instance, we need to first define a serialization schedule for it. The serialization schedule broadly contains two components, namely a static component and a dynamic component. The static component is defined with the help of analysis of the application task-graph, whereas the dynamic components are determined at runtime. These two components are discussed below:

8.7.1 Static Components to the Serialization Schedule

There are essentially two parts to the static components of the serialization schedule. They are:

- When composing over computation space, the non-blocking schedule for inner StepCapsule instances is required.
- For computations composed over iteration space, the computation spaces need to know about the dimensional boundaries at which all input edges lie with respect to their own iteration space dimensions.

Both these components are produced *a priori* at program start-up by analyzing the application task-graph. The non-blocking schedule is determined by resolving data-dependencies as expressed by the directed edges in the task-graph. The dimensional boundaries is produced by comparing the dimensional components of the iteration spaces of the producer/-consumer StepCapsule Space and the produced/consumed Item/Tag Capsule Space.

8.7.2 Dynamic Components to the Serialization Schedule

There are two components dynamically determined to specify the serialization schedule for every coarse-grain StepCapsule instance. These two conditions are:

Table 3: Schedules are called by examining the type and conditions set for a StepCapsule instance. Here, serialization is assumed to be *true* for all four cases.

Composed over Comp. Space	Iteration-Major	Computation Major
Yes	*executeFunctionCG_IM*	*executeFunctionCG_CM*
No	*executeFunctionFG_IM*	N/A

- Is the StepCapsule instance composed over computation space, and does it need to be serialized.
- Should the StepCapsule instance be serially executed in Iteration-Major mode or Computation-Major mode.

The code sample in Figure 28 shows how the two conditions are checked and the appropriate execution schedule selected.

```
1  bool StepCapsule_t::run(void) {
     bool retVal = false;
3    if(fineGrain == true) {
       retVal = executeFunctionFG_IM();
5    } else { // StepCapsule composed over Computation Space
       if(serialExe == true) {
7        if(space->isIterMajor())
           retVal = executeFunctionCG_IM();
9        else
           retVal = executeFunctionCG_CM();
11     } else { // (serialExe == false)
         // Run GC on this StepCapsule container
13       retVal = isEmpty();
       }
15   }
     return retVal;
17 }// run()
```

Figure 28: StepCapsule_t::run() function.

The top-level *run()* function is universally called to execute any StepCapsule instance. It checks first if the StepCapsule instance has been composed over computation space. If it is *not* a composed computation but a fine-grain user-defined function StepCapsule, the *executeFunctionFG_IM()* schedule is invoked. The *executeFunctionFG_IM()* executes

the instance in Iteration-Major mode, the only option available for fine-grain StepCapsule instances (Table 3). Otherwise, if the instance is composed over computation space with inner StepCapsule instances, the serialization condition *serialExe* is checked. The *serialExe* variable determines if the computation needs to be executed serially or, the composed computation is to act as a GC container. If the StepCapsule is a GC container, its GC condition is checked. Inner computations within a GC container continue to execute in parallel. If the StepCapsule instance is not a GC container, it is serialized. Again, for computations composed over computation space, there are two available serialization options, namely the Iteration-Major mode and the Computation-Major mode. The choice between the two serialization schedules is made dynamically at runtime and the appropriate schedule is invoked.

8.7.3 Execution Modes

In this section we describe the use of the following possible serialization schedules for composed computations in the Capsules programming model:

- executeFunctionFG_IM(); // in text
- executeFunctionCG_IM(); // in text
- executeFunctionCG_CM(); // in text

The functions below are components of the three serialization functions above and are called to perform the different tasks such as iterating through the TagCapsule instance tree, getting and unpacking coarse-grain ItemCapsule instance data, and packing and emitting coarse-grain Item/Tag Capsule instances.

- emitTagAndItemCapsules(); // in text
- autoSCFuncWithoutGet_IM(); // in appendix
- autoSCFuncWithGetFSE_IM(); // in appendix

- autoSCFuncWithGetFSEandPSE_CM(); // in appendix
- executeCG_SerialSchedule_IM(); // in text
- executeCG_SerialSchedule_CM(); // in appendix

8.7.3.1 Iteration-Major Serial Execution Order

```
1  bool StepCapsule_t::executeFunctionFG_IM(void) {
     initDataStructures(); // initialize environment
3    bool retVal = false;
     if(space->havePSIEs() == true) {
5      retVal = populateDataTreeFSE(ldt, ltc, 0, false);
       if(retVal != false) {
7        retVal = populateDataTreePSE(ldt, ltc, 0, false);
       }
9      if(retVal != false)
         retVal = autoSCFuncWithoutGet_IM(ldt,
11         ltc, 0, Tag_t::NullTag, Tag_t::NullTag);
       }
13   } else { // no PSIEs
       retVal = autoSCFuncWithGetFSE_IM(ldt,
15       ltc, 0, Tag_t::NullTag, Tag_t::NullTag, false);
     }
17   if(retVal != false) {
       emitTagAndItemCapsules(ltc);
19   }
     return retVal;
21 }// executeFunctionFG_IM()
```

1

Figure 29: executeFunctionFG_IM() function

```
1  bool StepCapsule_t::executeFunctionCG_IM(void) {
     bool retVal = populateCacheFSE(ltc, 0, false);
3    if(retVal != false) {
       retVal = executeCG_SerialSchedule_IM();
5    }
     if(retVal != false) {
7      emitTagAndItemCapsules(ltc);
     }
9    return retVal;
   }// executeFunctionCG_IM()
```

Figure 30: executeFunctionCG_IM() function

The source code in Figures 29, 30 and 31 illustrates how the pre-analyzed non-blocking schedule of the Computation-Space composed StepCapsule Space is used to execute a StepCapsule instance. The function uses the *schedule* list to determine the StepCapsule Pool to select and execute its instances. Note, all instances within a selected StepCapsule Pool must complete execution before proceeding to the next StepCapsule Pool. The same StepCapsule Pool is not revisited for execution. The execution schedule relies on the property that disallows cycles in the application task-graph. Further details regarding this property are explained in Section 3.6. The source code below executes the computation-space composed StepCapsule instance in Iteration-Major serialization mode. The serialization code that executes the computation space composed StepCapsule instance in Computation-Major mode is included for reference in appendix C.

```
bool StepCapsule_t::executeCG_SerialSchedule_IM(void) {
  bool retVal = false;
  for(int i = resumeIndex; i < static_cast<int>(schedule.size()); i++) {
    StepCapsuleSpace_t* scs = schedule[i];
    StepCapsulePool_t* scsPool = scPools[scs->getName()];
    retVal = scsPool->run();
    if(retVal == true) {
      assert(scsPool->isEmpty() == true);
    } else { // (retVal == false)
      resumeIndex = i;
      break;
    }
  }// for(len)
  return retVal;
}// executeCG_SerialSchedule_IM()
```

Figure 31: executeCG_SerialSchedule_IM() function

8.7.3.2 Computation-Major Serial Execution Order

Illustrated in Figure 32 is the source code to execute a Computation-Space composed StepCapsule in Computation-Major (CM) mode. The autoSCFuncWithGetFSEandPSE_CM() function unrolls the TagCapsule tree and executes the computation-space on the finest grain iteration along with getting data over both FSIEs and PSIEs.

```
1  bool StepCapsule_t::executeFunctionCG_CM(void) {
     initDataStructures(); // initialize environment
3    bool retVal = autoSCFuncWithGetFSEandPSE_CM(ldt,
       ltc, 0, Tag_t::NullTag, Tag_t::NullTag, false);
5    if(retVal != false) {
       emitTagAndItemCapsules(ltc);
7    }
     return retVal;
9  }// executeFunctionCG_CM()
```

Figure 32: executeFunctionCG_CM() function

8.7.4 Getting Data at Coarse-Grain Computation Boundary

One of the key optimizations enabled by the Capsules Programming Model is the ability to aggregate data requests (or input synchronization points) to the coarse-grain computation boundary. These coarse-grain data requests are based on the dimensional relationships between the iteration spaces of the consuming computation and the data.

8.7.4.1 Data over FSIE

Retrieving data over a FSIE is straight forward as all dimensions of the data can be specified by inspecting the dimensions of the consuming computation.

For user-defined function StepCapsules, data over FSIEs is retrieved at the dimensional boundary during the unfolding of the TagCapsule tree. Fetching data at the dimensional boundary reduces the synchronization points required to the ItemCapsule Pool (Section 4.4).

If a computation space composed StepCapsule Space has an FSIE, the data over the edge is cached before initiating coarse-grain StepCapsule instance execution. Caching is done for both Computation-Major and Iteration-Major execution modes. Caching over FSIEs for computation space composed StepCapsules helps reduce synchronization points by coalescing multiple accesses by consumers of the ItemCapsule Space (if they exist).

8.7.4.2 Data over PSIE

As described in Section 4.9, PSIEs have the property of performing a dimensional reduction on the data's iteration space. However, for dimensional reduction to occur, the consuming computation must have *dimension definition functions* that specify the instance values of the reduced dimensions.

User-defined StepCapsule Spaces have these dimension definition functions provided for them by the user. The runtime system can therefore use them to retrieve the data for the user-defined StepCapsule Space.

Retrieving data over PSIEs into computation-space composed StepCapsule Spaces however is a bit tricky. Data over PSIEs into computation space StepCapsule Spaces has to be *lazily* gotten by the inner user-defined function StepCapsule Spaces that the edges terminate into. The reason for this is as follows: As described earlier, for dimensional reduction to occur, the consuming computation must have *dimension definition functions*. These functions, however, are only provided for the user-defined function StepCapsule Spaces and cannot be deduced for the Computation Space StepCapsule spaces.

Moreover, since a PSIE into computation space composed StepCapsules terminates into a user defined StepCapsule Space as either an FSIE, or a PSIE, the dimensions of the data can eventually become specified. In either case, data can only be lazily retrieved when executing the user-defined function StepCapsules. The lazy retrieval still enables data caching at the computation space composition boundary similar to that done for FSIEs.

8.7.5 Outputting Data at Coarse-Grain Computation Boundary

Once a composed coarse-grain StepCapsules have executed, their stored output in the Environment is packed and emitted to their respective pools in the runtime. The source code in Figure 33 illustrates the *emitTagAndItemCapsules* function, which is executed at the last dimensional boundary of the executing StepCapsule Space. The *emitTagAndItemCapsules* function emits data over all FSOEs and PSOEs.

```
1 void StepCapsule_t::emitTagAndItemCapsules(const TagCapsule_t& tc) {
    emitFSE_outItem(tc);
3   emitFSE_outTag(tc);
    emitPSE_outItem(tc);
5   emitPSE_outTag(tc);
  }// emitTagAndItemCapsules()
```

Figure 33: emitTagAndItemCapsules() function

CHAPTER IX

PERFORMANCE EVALUATION

In this chapter we describe our evaluation methodology, test cases and results for using the Capsules Parallel Programming Model with real and synthetic applications.

9.1 Evaluation Methodology

9.1.1 Metrics

The metrics used to evaluate the application performance and the underlying Capsule mechanisms are as follows:

The **Normalized Execution Time** is the ratio of the parallel execution time with respect to the serial execution time of the original unmodified application. Therefore, the normalized execution time is:

$$T_{Norm} = T_{Parallelized}/T_{Serial}$$

.

The **Total Work** metric represents the total amount of CPU time spent in both the application and the Capsules runtime on all available PEs. The metric helps determine how the entire parallel execution load behaves as the execution granularity is altered. The total work metric is measured in terms of OProfile samples (Section 9.1.2), which is directly proportional to wall-clock time:

$$Samples_{TotalWork} = |Samples_{application}| + |Samples_{runtime}|$$

The **Percentage Overhead** represents the fraction of the CPU time spent in the Capsules runtime with respect to the total work done on all PEs. The runtime overhead consists of all non-application related operations performed by the system. The overhead percentage

represents the efficiency with which the Capsules programming model is able to execute the application in parallel. Just like the *Total Work* metric, Percentage Overhead is also computed using OProfile [46] samples:

$$\%Overhead = |Samples_{runtime}|/|Samples_{TotalWork}|$$

.

The **Queue Imbalance** metric represents the standard deviation (SD) between the time each queue thread is active in executing work. The queue imbalance is a metric to measure the load imbalance in the processing elements over the execution of the entire program. The queue-imbalance QI can be mathematically represented as the unbiased standard deviation estimator for the active time for each queue thread in the parallel execution:

$$QI = \sqrt{(E(X) - X)^2/(N-1)}$$

In the equation above, X is the active time for a given queue thread, $E(X)$ is the expected mean of the active time for all queue threads, and N is the number of queue threads participating in the parallel execution.

9.1.2 Timing Infrastructure

The OProfile [46] statistical sampling tool was used to measure the runtime overhead. Samples gathered in application functions were separated from samples captured in the runtime using generic scripts that analyzed the profile to compute the application percentage overhead. Samples were captured at every 90K clock cycle intervals with a call-graph depth size of 10.

A timing infrastructure build for *a prior* work [25] to capture discrete events of interest during program execution was used. The higher cost of this timing infrastructure due is to its use of the *gettimeofday()* system call, and the extremely fine-grain nature of computations being measured required the timing infrastructure be used only sparingly. Specifically, this event infrastructure was used to measure the execution queue-imbalance.

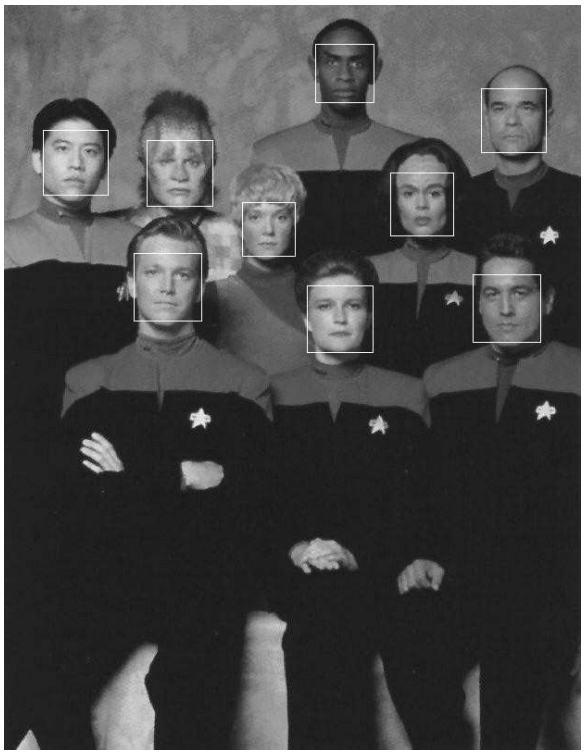


Figure 34: Cascade Face Detector: Result image with detected faces. Image from the MIT/CMU database [15, 73]

9.1.3 Hardware Platform

The hardware platform used for experimentation is a 2-Way SMP machine with two 1.6GHz Intel Core 2 Quad Clovertown processors giving a total of eight processing cores. The machine is also configured with 4GB RAM. A 32bit Fedora Core 6 Linux OS is used with the latest updated kernel version 2.6.22-14-72.

9.2 Applications

The following applications were parallelized using Capsules, namely (1) The Cascade Face Detector (FD) [77] (Figure 4), (2) a Stereo Vision depth (SV) algorithm [83] (Figure 2), a Synthetic N-stage Pipeline (SN) application (Figure 36) and (4) the Apply Filters kernel (AF).

9.2.1 Cascade Face Detector

The face detector applies a cascade of pre-computed simple facial features on a fixed size window within the image to detect the existence of a face in that window. The detection process is performed over the entire input image to detect faces on different locations by shifting the window by 1 pixel over the *X* and successively over the *Y* axes of the image. In order to detect faces of different sizes, the image is scaled down and the detection process is repeated again. The scaling and re-detection process is repeated until the image scales down to the size of the feature detection window. A post processing phase, takes all detected faces and prunes duplicates that are close to each other. The final list of detected faces is then marked into a copy of the input image as illustrated in Figure 34. The images used by the face detector are from the MIT/CMU combined frontal face database [15, 73].

9.2.2 Stereo Vision Depth

The Stereo Vision Depth algorithm (SV), also known as the Stereo Correspondence algorithm, is used to detect how far objects are placed in a given scene (Figure 35). It outputs a depth map from two stereo input images. The first stage in the algorithm consists of building multiple disparity-maps for the two input images. The second stage re-samples the disparity-maps to find the highest disparity values. The final stage composes the re-sampled disparity-maps to create the final depth-map. The input image used is from the work described in [72].

9.2.3 Synthetic N-Stage Pipeline Application

A synthetic N-stage pipeline application was constructed to illustrate the reduction in overhead when composing over computation space. The application consists of three StepCapsule Spaces called *Start*, *Baz* and *End*. StepCapsules *Start* and *End* are fine-grain user-defined steppers, whereas *Baz* is a coarse-grain StepCapsule Space composed of a pipeline of N fine-grain *Baz_N* StepCapsule Spaces. Each fine grain *Baz_N* takes in two inputs. The

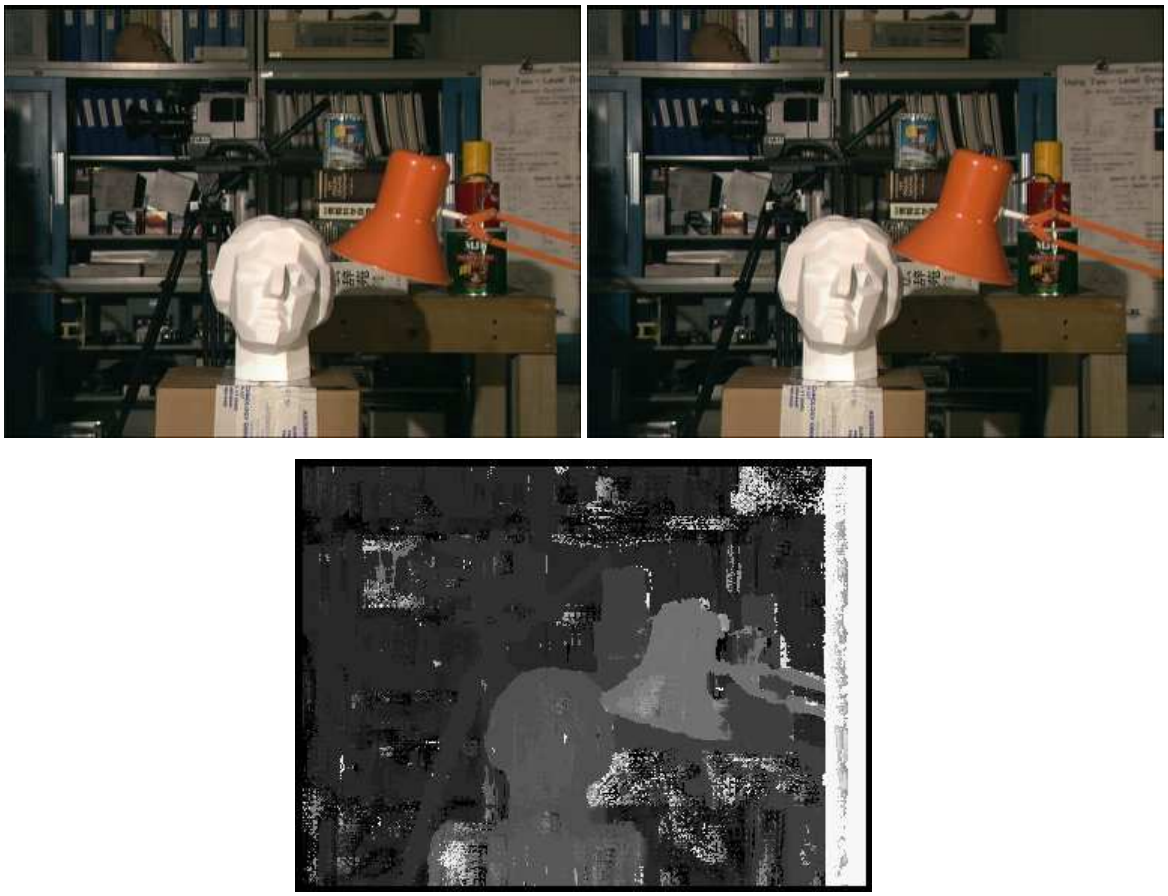


Figure 35: Stereo Vision Depth: Left and right input stereo images [72] and result depth map

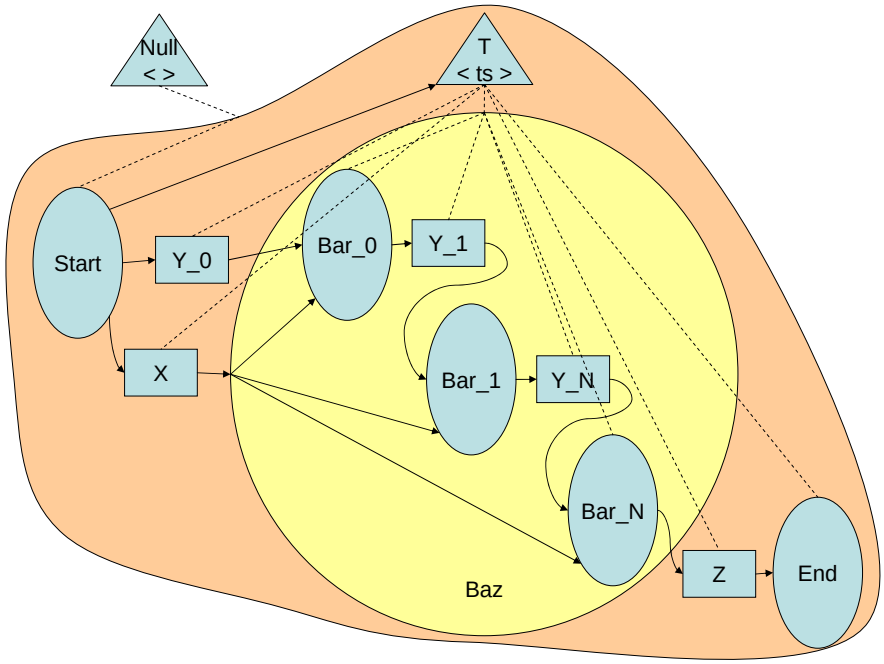


Figure 36: Synthetic N-Stage Pipeline Application task-graph in Capsules

first input is the item *X* produced by *Start*. The second input is the item *Y_(N-1)* produced by *Baz_(N-1)*, the previous StepCapsule in the pipeline. Notice that in the composed coarse-grain StepCapsule *Baz*, all *N* input synchronization points from *X* to *Bar_N* are merged into *one* synchronization point. The merged synchronization point is illustrated by the single input edge drawn from *X* to *Baz*.

It is interesting to note that individual *Bar_N* finest-grain StepCapsule Spaces can be composed into sets of coarser-grain StepCapsule Spaces inside *Baz*. A programmer may want to create such compositions in a pipeline with stages of varying computation time. Stages that are too fine-grain can be composed together to create coarser-grain stages that are balanced in execution time with respect to each other. Composing and serializing some of the *Bar_N* stages would also reduce overhead.

The synthetic N-stage application task-graph illustrates a commonly found trend in vision applications where multiple stages of a computation pipeline need access to the same data. For example, the Color People Tracker application used in investigating programming models for dynamic vision applications [65, 64] also fits into this task-graph

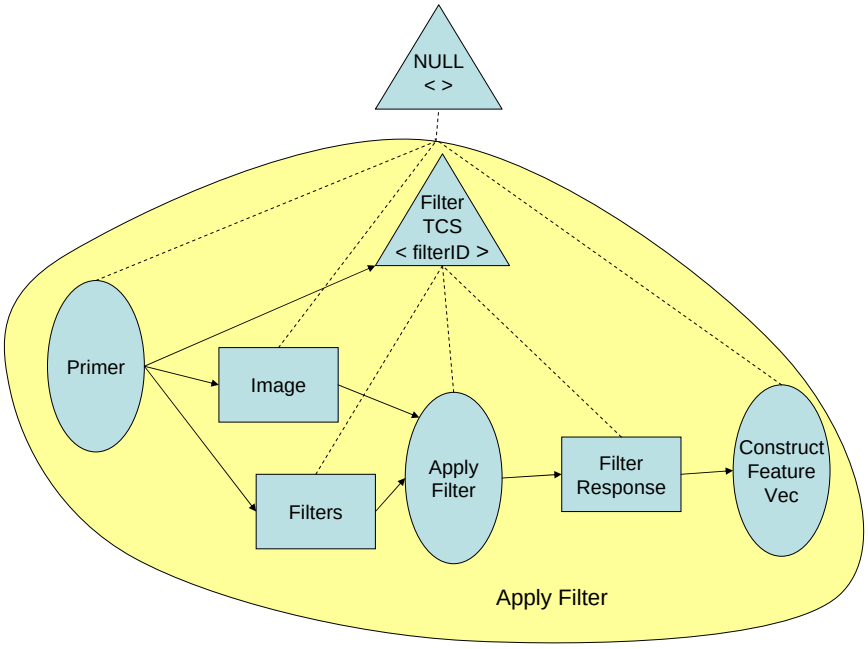


Figure 37: Apply Filters task-graph in Capsules

pattern. In the tracker application, the stages Background Subtraction, Histogram Equalization, and even the final Tracking Stage require the same image as input. In such cases, composition over computation space can help increase data locality for all pipeline stage computations. Moreso, serializing the composed coarse-grain computation helps reduce synchronization points and book-keeping overheads incurred to maintain data-dependency. Data-dependency resolution costs could include blocking computations due to unavailable data, and restarting or resuming them later once the data became available. The results analysis shows that such overhead reduction patterns are also seen in the Stereo Vision Depth application (Section 9.4).

9.2.4 Apply Filters kernel

The *Apply Filters kernel* (AF) illustrated in Figure 37 is a generic application kernel used to apply a set of filters on an image using convolution to produce a filter response. The filter responses are then reconstructed into a feature vector that may be used in different ways depending on the application. The Apply Filters kernel is widely used in machine vision

applications, and in our case, it is part of a live robot path planning algorithm that uses a set of pre-computed feature filters to determine the best path to take given input images of the robot's field of view. The robot path planning work described is part of Georgia Tech's Learning Applied to Ground Robots (LAGR) project [18].

The experiments below were performed with input images of size *128 x 96* pixels and with *96* filters each of size *21 x 21* pixels.

9.3 Results

9.3.1 Expectations

By investigating the metrics in Section 9.1.1 for the applications described earlier, we hope to support the hypothesis that granularity control can be useful for applications to optimize their parallel execution. In particular, we hope to show that increasing granularity helps reduce the parallelization overheads and improve application performance. Moreover, we hope to show that granularity should be increased with care, such that an execution with computations too coarse-grain would lead to a loss in parallelism causing a load imbalance in the runtime.

Specifically, we hope to show that both composition over iteration space and composition over computation space can be useful in decreasing the parallelization overhead. The reduction in overhead can be verified using the percentage overhead and the total work metrics. Both metrics expect to decrease in magnitude when the execution granularity is increased. Moreover, we expect the application performance to improve when increasing granularity, which we plan to show using the Normalized Execution Time metric. The Normalized Execution Time can also indicate when the granularity has become too coarse grain such that the application performance actually decreases with increased granularity.

The Queue Imbalance metric is another indicator to verify when coarse-grain computations are causing a load imbalance. We expect the Queue Imbalance to remain low with

fine-grain computations and increase when the granularity over iteration space or computation space get too coarse-grain.

We note that since most applications have greater potential parallelism expressed over iteration space rather than over computation space (via task parallelism and pipelining), we expect better gains in performance when composing over iteration space than when composing over computation space. Likewise, if applications do not have enough parallelism over iteration space either, we expect no performance improvement by increasing the granularity over iteration space. Given input images of sufficient size, applications such as the Cascade Face Detector (FD) and Stereo Vision Depth (SV) algorithms exhibit more parallelism over iteration space than over computation space. For the Synthetic N-Stage application, we compose only over computation space and not over iteration space to verify the reduction in overheads by the increasing granularity over computation space not expected to be shown by other applications.

Furthermore, when both compositions are used together to create even coarser-grain compositions, we expect the performance to improve further. However, in the application test cases investigated, only the FD, SV and N-Pipe applications provide an opportunity to compose over both computation space and iteration space. The N-Pipe as explained earlier, is specifically used to investigate the performance benefits of composition over computation space. The FD application if composed over computation space would serialize the face detection on the entire image and therefore composition over computation space cannot be enabled. Therefore, only the SV application is left to investigate dual compositions. For SV, since there are only two unique StepCapsule Spaces that can be composed together over computation space to form the coarse-grain *Disparity Computation* StepCapsule Space, there again is not enough granularity increase to show an improvement in performance.

Another dimension to investigate is the changing availability of processing resources (or PEs) and the impact of granularity adjustment in adapting to those changes. We expect that parallel executions are sensitive to resource change and that having the wrong

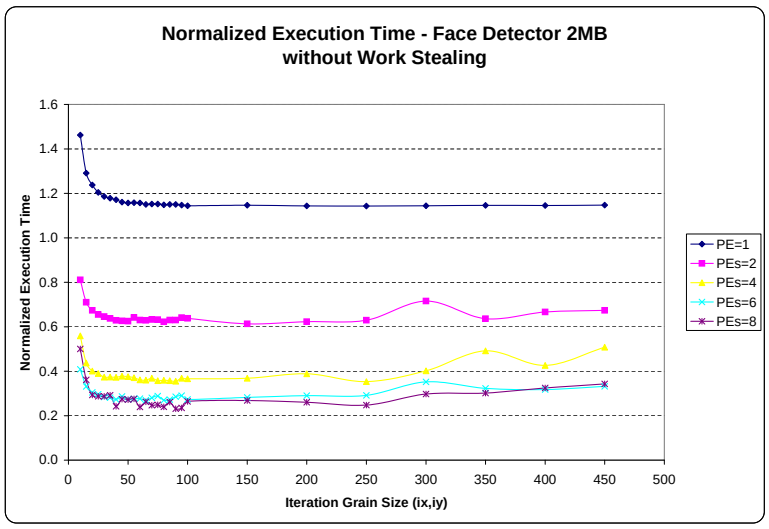


Figure 38: Normalized Execution Time; Cascade Face Detector; x-axis: Granularity for dimensions (ix, iy); y-axis: Normalized Execution Time.

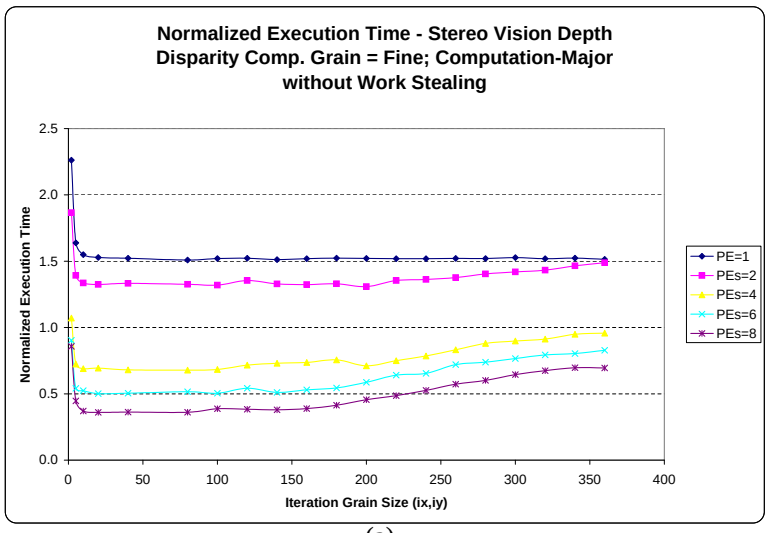
granularity for a given number of available resources could adversely affect performance. Therefore, a high granularity may be beneficial for performance and reducing overhead when fewer concurrent hardware resources are available. However, when resource availability increases, the coarse-grain computations would not improve performance due to an under-utilization of the hardware concurrency.

9.3.2 Normalized Execution Time

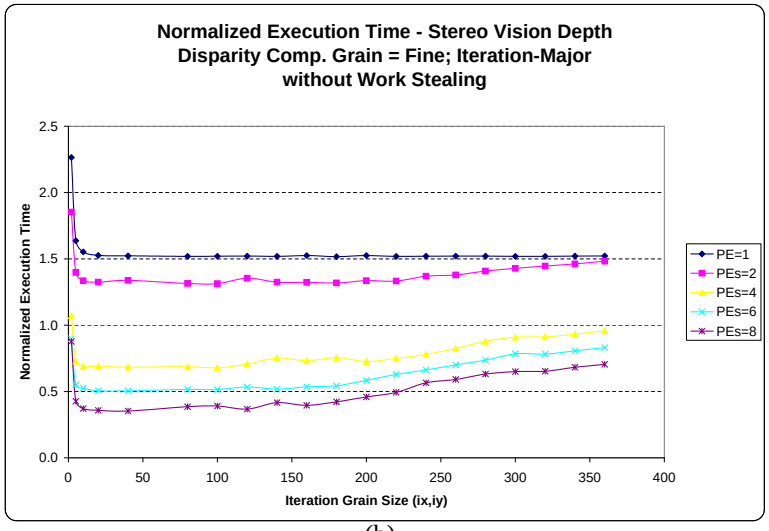
Figures 38, 39, 40 and 41 illustrate the *normalized execution time* of the applications on different number of available PEs.

The x-axis for the FD graph (Figure 38) represents the granularity selected for the detection windows in both the x and y axis location in the image $< ix, iy >$. The total number of detection windows grouped together to form a coarse-grain computation is therefore equal to the square of the granularity selected.

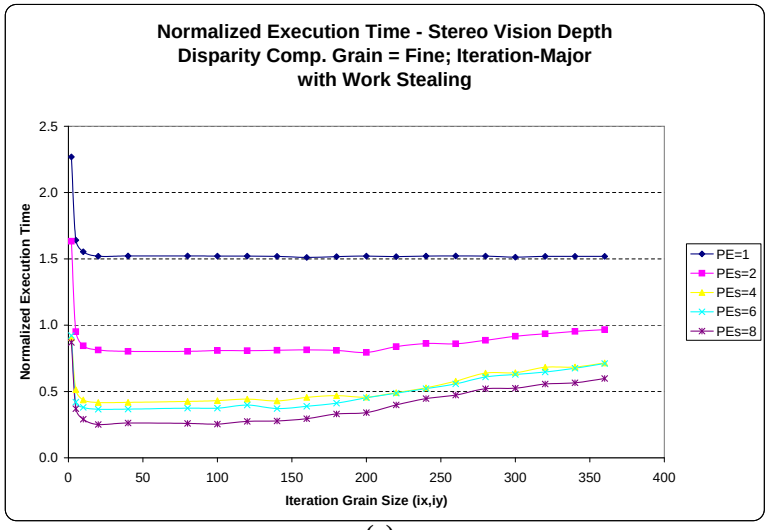
The x-axis for SV graphs (Figures 39, 40) also represents the granularity selected for dimensions $< ix, iy >$. These two dimensions are again the x and y pixel locations that are grouped together for the first phase of processing. There is a third iteration dimension called $disparity$ (application task-graph in Figure 2), which represents the disparity-maps



(a)

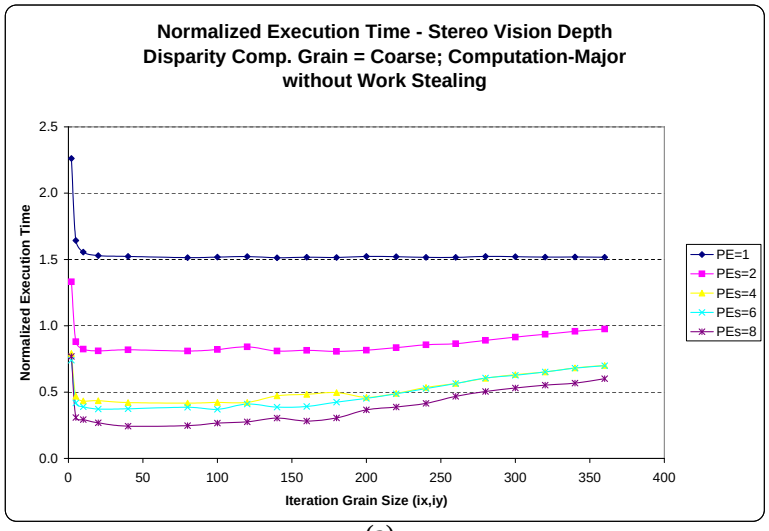


(b)

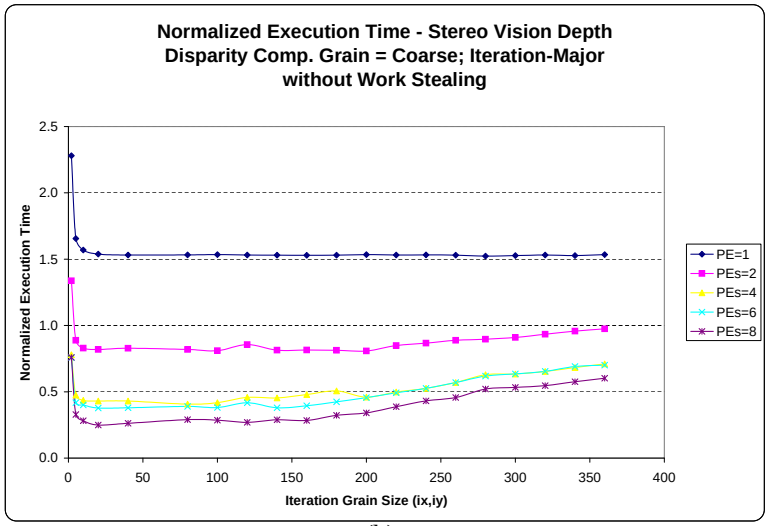


(c)

Figure 39: Normalized Execution Time; Stereo Vision Depth with Disparity Computation Grain - Fine; The last graph represents Work Stealing enabled runtime. x-axis; Granularity for dimensions (x,y); y-axis: Normalized Execution Time.



(a)



(b)

Figure 40: Normalized Execution Time; Stereo Vision Depth with Disparity Computation Grain - Coarse; x-axis; Granularity for dimensions (x,y); y-axis: Normalized Execution Time.

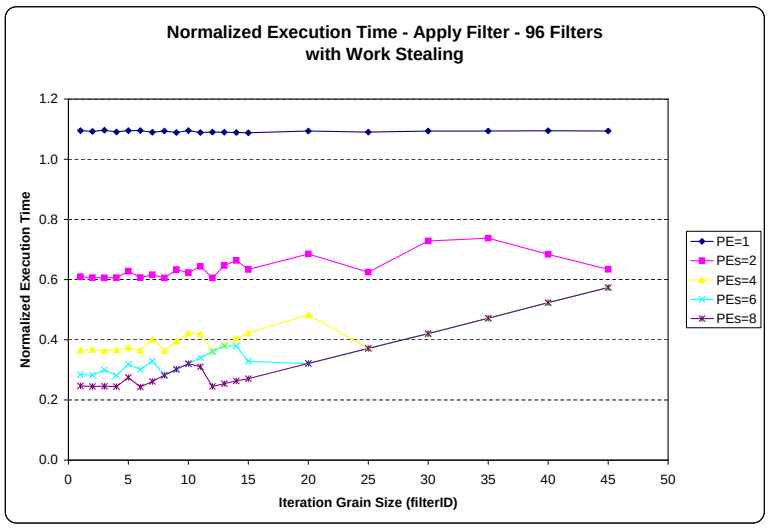


Figure 41: Normalized Execution Time; Apply Filters x-axis; Granularity for dimension (filterID); y-axis: Normalized Execution Time.

grouped together for processing of the second phase of the algorithm. The granularity for disparity was fixed at 4 for our experimentation. Varying the granularity of disparity also affects application performance just like it does when varying the grain size for other dimensions $< ix, iy >$. Since the overall trends were the same, we exclude adjusting the granularity of $disparity$ from our main analysis for brevity and simplification of data visualization.

Note that there are five graphs illustrating SV results. The graphs in Figures 39(a) and 39(b) represent the SV application executing *without* serialization of the *Disparity Computation* StepCapsule Space. The *Disparity Computation* is composed over computation-space using the finer-grain *Build Disparity Image* and *Resample Disparity Image* StepCapsule Spaces. The graph in Figure 39(c) represents Work-Stealing [51] enabled in the Capsules runtime. The use of Work-Stealing is explained in Section 9.4.

The graphs in Figures 40(a) and 40(b) illustrate SV results *with* the serialization of the composed *Disparity Computation* StepCapsule Space. These experiments show how composition over computation space and its successive serialization helps reduce overhead and improve application performance.

The x-axis for the AF graph (Figure 41) represents the granularity selected for the

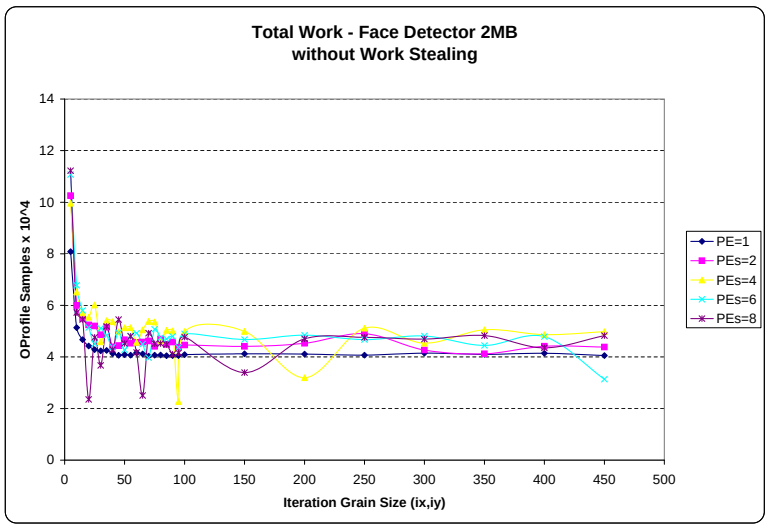


Figure 42: Total Work: Cascade Face Detector; x-axis: Granularity for dimensions (x, y); y-axis: Number of Samples captured on all PEs

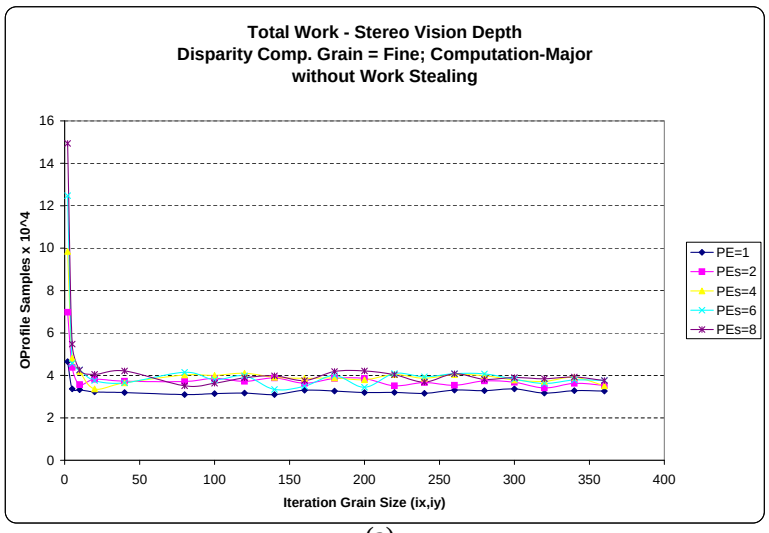
$filterID$ dimension. Composition over the $filterID$ dimension represents coarse-grain StepCapsules that perform multiple *apply filter* operations on the image atomically.

9.3.3 Total Work

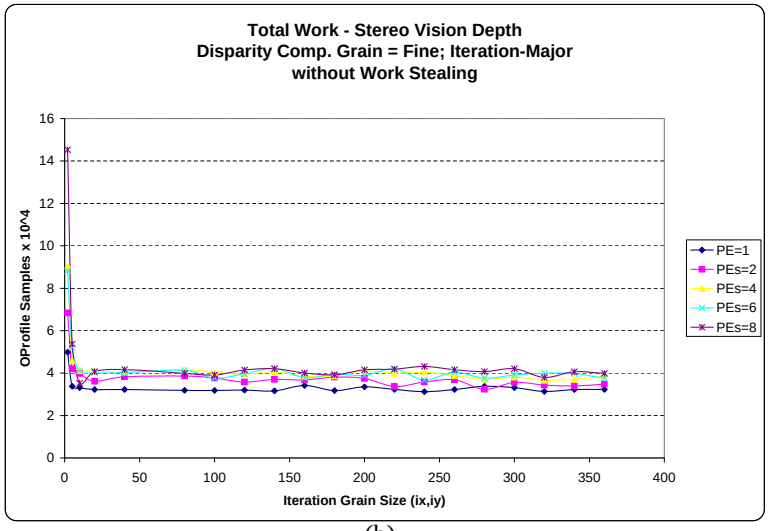
Figures 42, 43 and 44 illustrate the *Total Work* done on all PEs for both applications. The x-axis represents granularity as described earlier in Section 9.3.2. The y-axis represents OProfile samples captured on all PEs. The number of samples captured by OProfile is directly proportional to the time spent on a given CPU.

9.3.4 Percentage Overhead

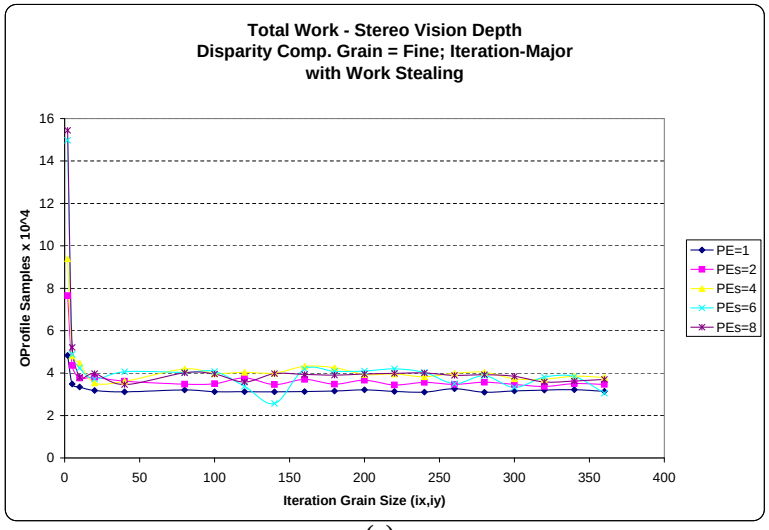
Figures 45, 46 and 47 illustrate the *percentage overhead* of the Capsules runtime mechanisms with respect to the Total Work done for a given application execution. The percentage overhead metric supports the hypothesis that creating coarse-grain computations and executing them serially in the absence of hardware parallelism reduces parallelization overhead and improves overall application performance. The x-axis in the graphs denotes increasing iteration space granularity for both applications FD and SV as described in Section 9.3.2. The y-axis denotes the percentage overhead as described in Section 9.1.1.



(a)

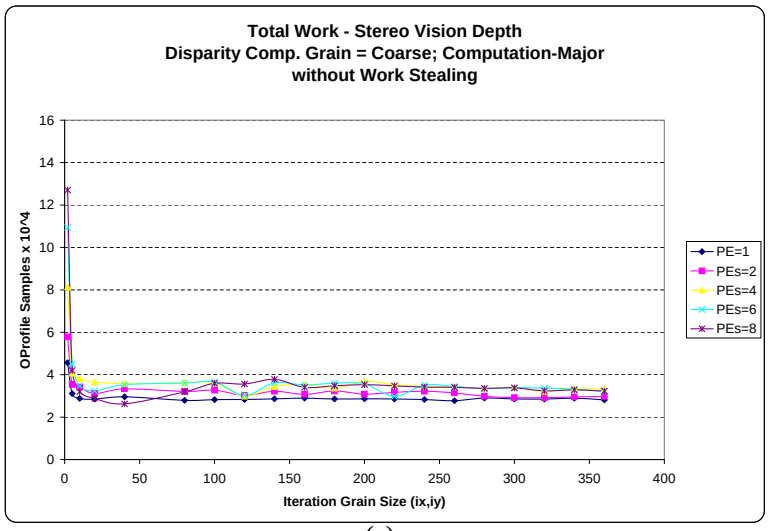


(b)

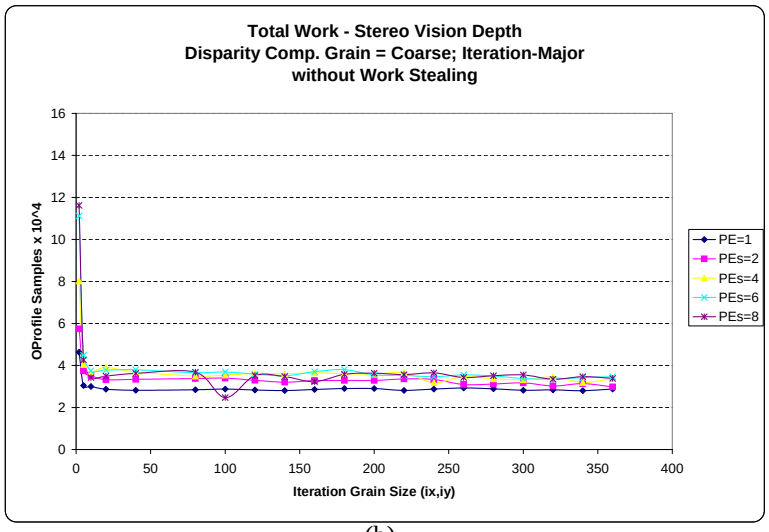


(c)

Figure 43: Total Work: Stereo Vision Depth with Disparity Computation Grain - Fine; The last graph represents Work Stealing enabled runtime. x -axis: Granularity for dimensions (x, y); y-axis: Number of Samples captured on all PEs



(a)



(b)

Figure 44: Total Work: Stereo Vision Depth with Disparity Computation Grain - Coarse; x -axis: Granularity for dimensions (x, y); y-axis: Number of Samples captured on all PEs

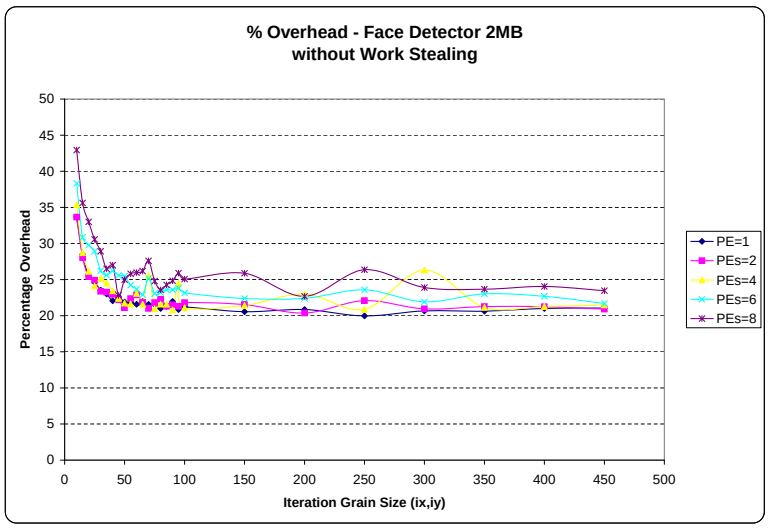


Figure 45: Percentage Overhead: Cascade Face Detector; x-axis: Increasing Granularity; y-axis: Percentage Overhead

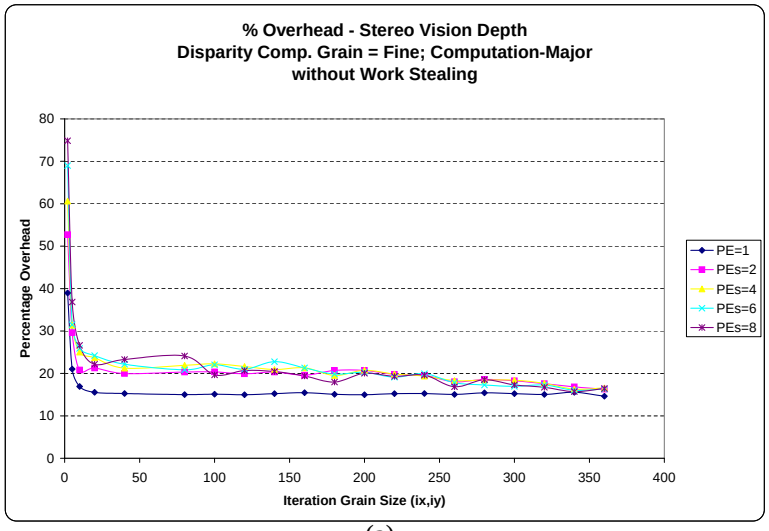
9.3.5 Queue Imbalance

Figures 48 and 49 illustrate the Queue Imbalance specifically measured for the SV application. The Queue Imbalance is used to explain the difference in speedup in the Normalized Execution Times for greater than 1 PE experiments when serializing the composed StepCapsule Space *Disparity Computation* verses not serializing the composed *Disparity Computation*. More details about SV are given in Section 9.4.

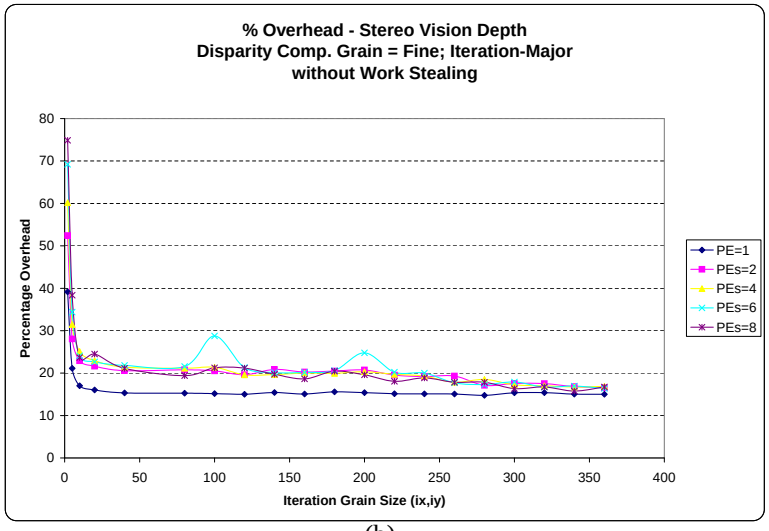
Figure 50 illustrates the Queue Imbalance for the AF kernel with Work-Stealing enabled. The graph shows a low imbalance during execution, indicating a high utilization of the concurrent hardware especially when granularity over iteration space is low.

9.3.6 Reducing Overhead when Composing over Computation Space

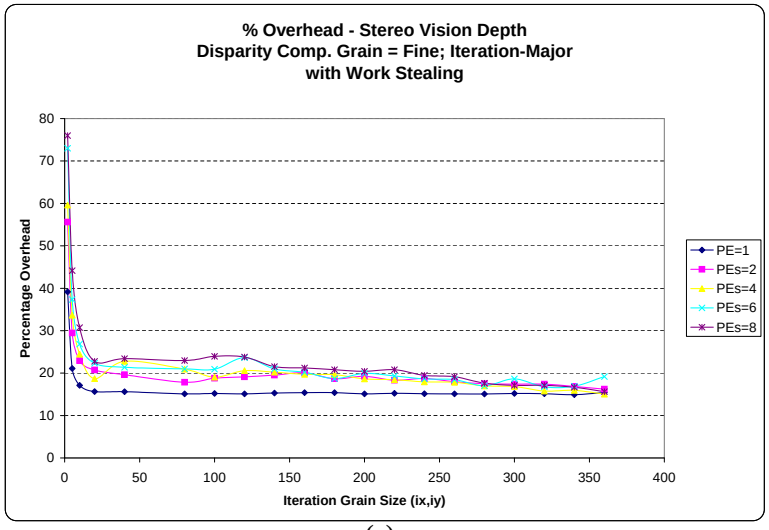
Figure 51 illustrates overhead incurred in the Synthetic N-Stage application task-graph described in Figure 36. Since this application does no work inside the finest-grain functions, the application execution time denotes pure overhead incurred in the Capsules runtime. The x-axis represents N, the number of stages in the pipeline computation represented by the coarse-grain StepCapsule Space labeled *baz*. The y-axis represents execution time for 10,000 iterations on the $< ts >$ dimension in the task-graph (Figure 36).



(a)

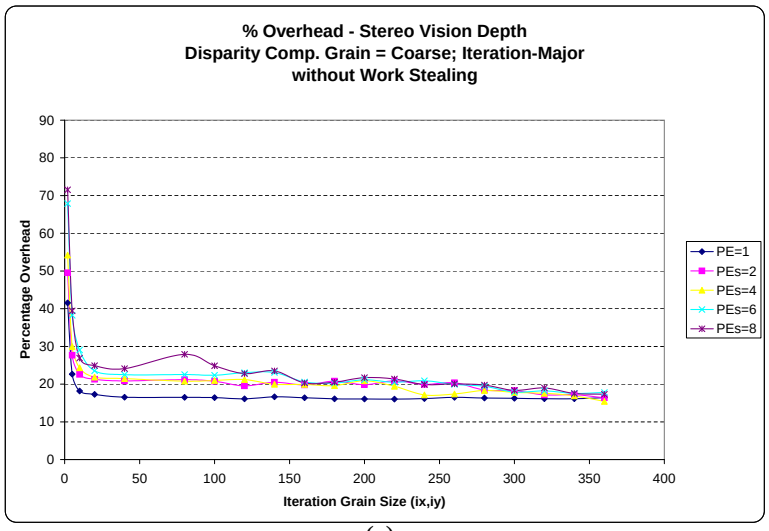


(b)

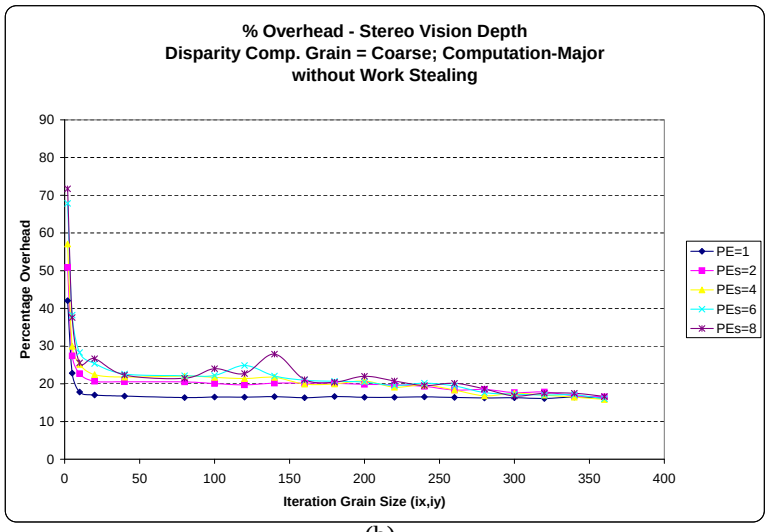


(c)

Figure 46: Percentage Overhead: Stereo Vision Depth with Disparity Computation Grain - Fine; The last graph represents Work Stealing enabled runtime. x-axis: Granularity for dimensions (x, y); y-axis: Percentage Overhead

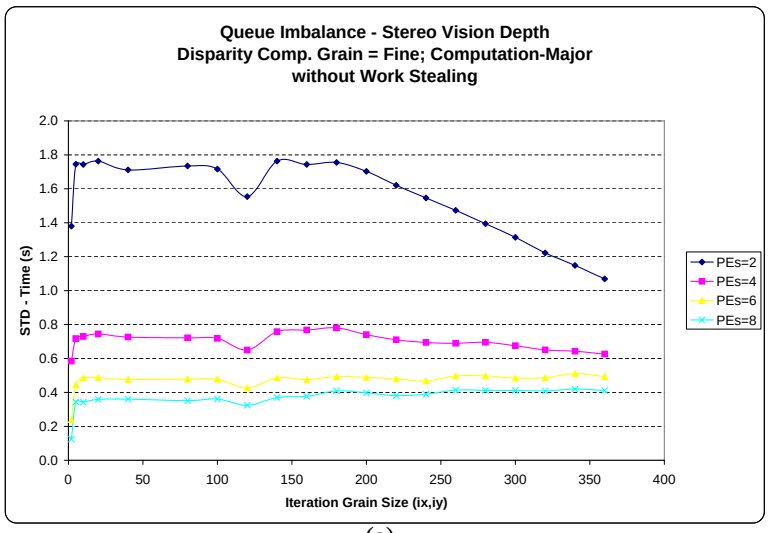


(a)

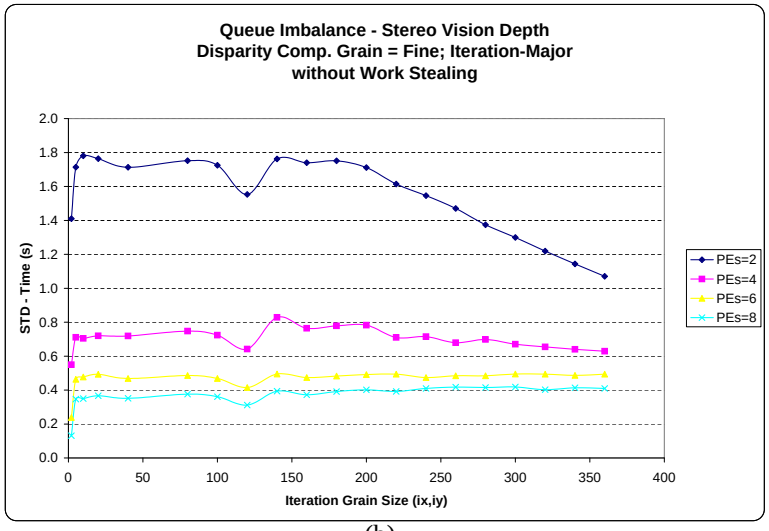


(b)

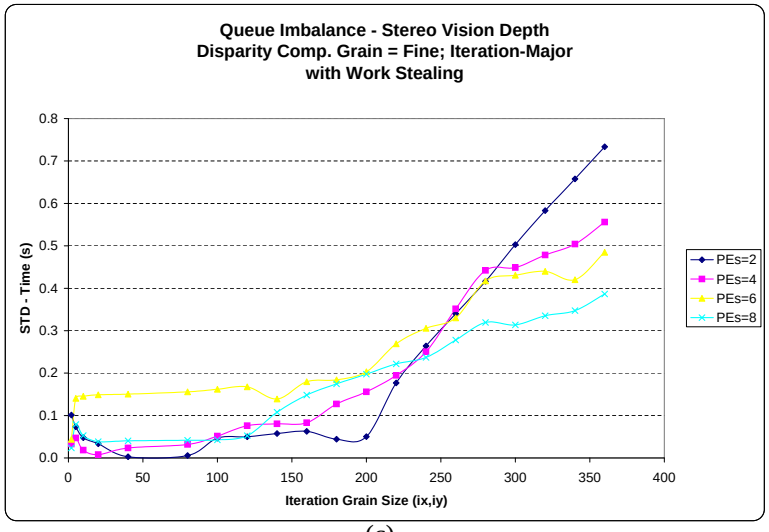
Figure 47: Percentage Overhead: Stereo Vision Depth with Disparity Computation Grain - Coarse; x-axis: Granularity for dimensions (x, y); y-axis: Percentage Overhead



(a)

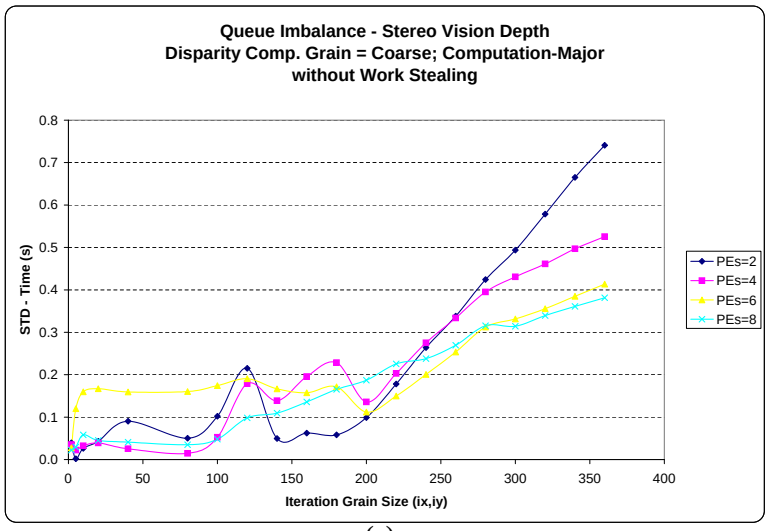


(b)

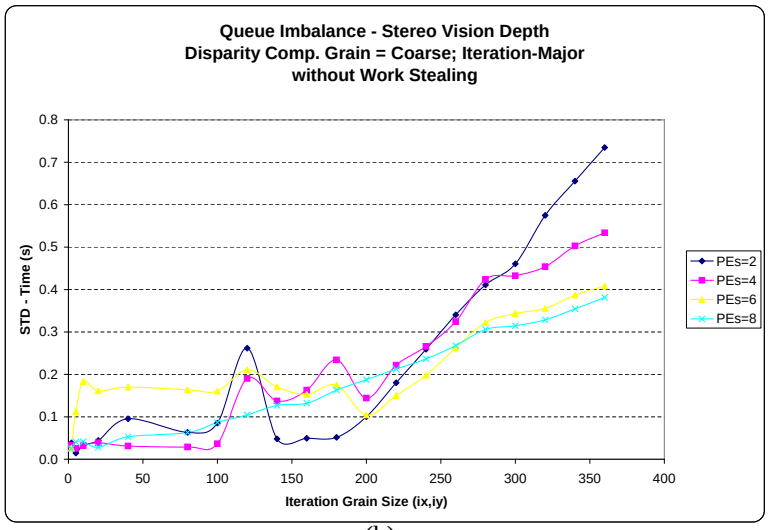


(c)

Figure 48: Queue Imbalance: Stereo Vision Depth with Disparity Computation Grain - Fine; Graph 48(c) represents Work Stealing enabled. x -axis: Granularity for dimensions (x,y); y-axis: STD (s)



(a)



(b)

Figure 49: Queue Imbalance: Stereo Vision Depth with Disparity Computation Grain - Coarse; x -axis: Granularity for dimensions (x, y); y-axis: STD (s)

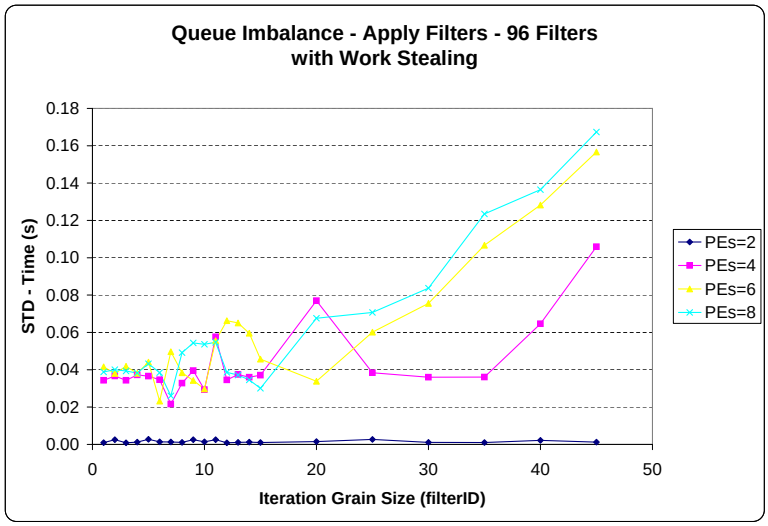


Figure 50: Queue Imbalance: Apply Filters; x -axis: Granularity for dimension (filterIDs); y-axis: STD (s)

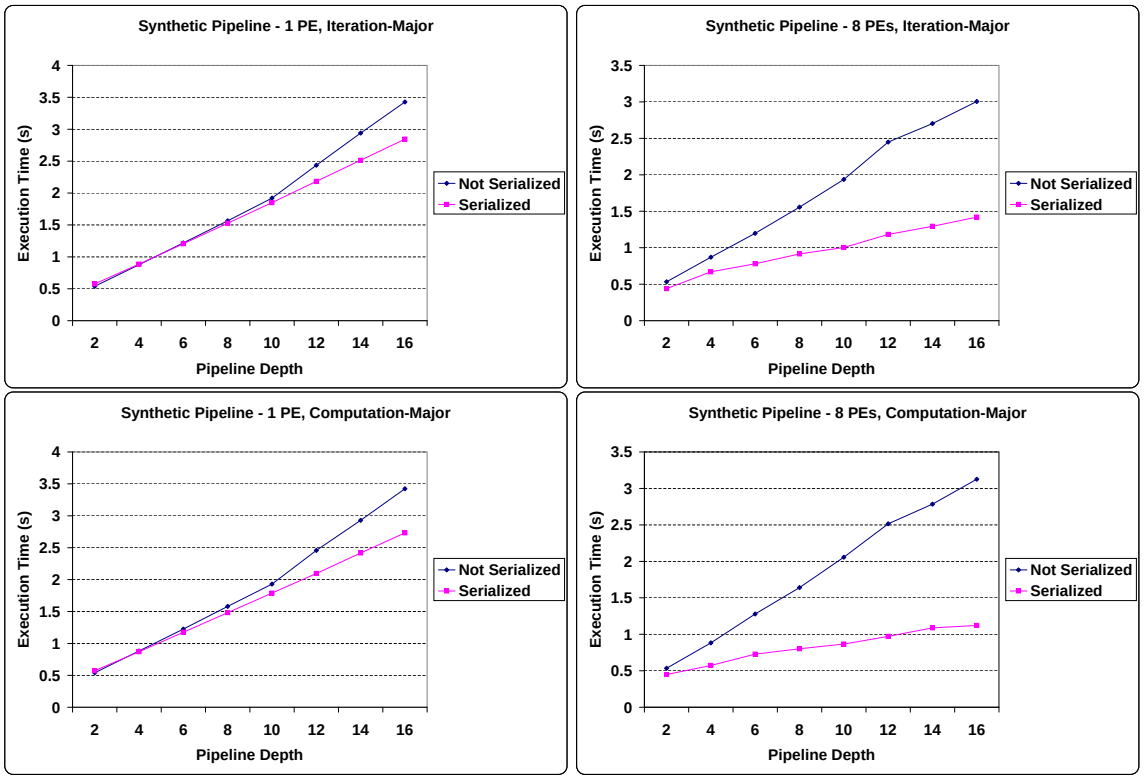


Figure 51: Overhead in Synthetic Pipeline Application: Execution Mode, (top) Iteration-Major (1,8 PEs) and (bottom) Computation-Major (1,8 PEs); x-axis: Execution Time; y-axis: Pipeline Depth

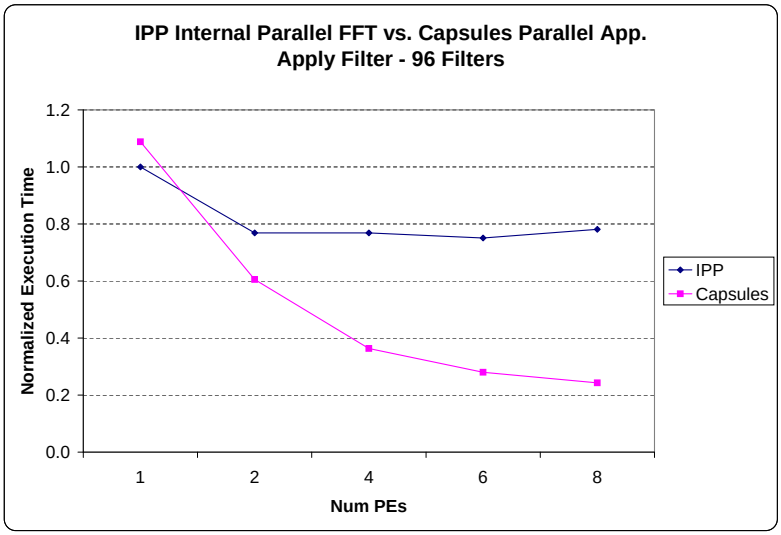


Figure 52: Apply Filters kernel comparison: IPP and fine-grain parallelism vs. Capsules and coarse-grain parallelism; x-axis: Number of PEs; y-axis: Normalized Execution Time

9.3.7 Apply Filter Comparison; Capsules vs. IPP

As described in Section 9.2.4, the Apply Filters kernel uses convolution to apply a number of filters to an input image. The convolution in this case is performed using the Intel Performance Primitive (IPP) library's convolution function. Interestingly, IPP also provides an internal FFT-based parallel implementation that can be invoked to utilize multiple hardware cores when available. We compare the results of using IPP's internal fine-grain parallelism in a serial for-loop iterating through all filters verses using coarse-grain parallel execution at the filter level using Capsules, and executing each IPP convolution (apply filter) serially. The experiments illustrate the trade-off between using fine-grain parallelism with barriers or coarse-grain parallelism without barriers. The results for this experiment are shown in Figure 52. The graph is analyzed in detail in Section 9.4.

9.4 Results Analysis

Figures 38 and 39 illustrating the Normalized Execution Time for FD and SV clearly show that increasing the granularity over iteration space increases the performance of the application's parallel execution. The results also show that the rate of execution speedup denoted

by the graph gradient, is initially high but falls quickly to zero, before becoming negative for large granularity values. The graph trend shows that increasing the granularity initially makes a large impact on application performance speed-up, but begins to have a lower impact when the granularity becomes larger and larger with eventually having a detrimental effect of reducing speedup. Such a behavior is expected as discussed in Section 9.3.1, in that increasing granularity helps reduce the parallelization overhead but at the same time reduces application parallelism. Therefore, increasing the granularity serializes the application and reduces its ability to execute in parallel thereby restricting its theoretical speedup limit (Section 2.5). As the granularity keeps increasing, the Normalized Execution Time graph begins to level-out back to a zero gradient. At this point, the application parallelism matches well with the available hardware concurrency. It can also be noted that there is a large granularity range where there is minimal change in application speedup. The range over iteration space where the graph lines remain flat is noted as the *Low Impact Iteration Space Granularity Range* (LIISGR). The graphs with this classic *bath tub* shaped result is also illustrated by previous works when investigating the impact of granularity [34, 29] on applications.

The graph in Figure 39(c) illustrates the Normalized Execution Time for the SV application with work-stealing enabled in the runtime. Investigation reveals that the default round-robin distribution scheme of the Capsules runtime groups and assigns the *Build Disparity Image* StepCapsules instances and its data-dependent *Resample Disparity Image* StepCapsule instances into disjoint execution queues. The disjoint distribution of StepCapsule instances cause work queues with the data-dependent *Resample Disparity Image* instances to block all its computations for unavailable data. Blocked computations lead to some work queues without available tasks, thereby creating a load imbalance in the system. The load imbalance is visible by observing the Queue Imbalance graphs *without* Work-Stealing enabled in Figures 48(a) and 48(b). The graphs have a sharply increasing Queue Imbalance for all PE experiments, that remains high with increasing granularity. To

resolve the load imbalance we have enabled work-stealing in the runtime that improves performance for 2+ PE experiments as can be seen between graphs in Figures 39(a), 39(b) (without work-stealing) and the graph in Figure 39(c) (with work-stealing).

In Figure 40, the granularity for *Disparity Computation* StepCapsule Space in the SV application is set to coarse (*i.e.,* composed and serialized). It can be noted that with respect to the finer-grain execution of *Disparity Computation* (composed but not serialized) in Figure 39, the application displays considerable speedup for the coarse-grain execution in all but the 1 PE experiment. Such a speedup is partly due to the reduced overhead due to serial execution of composed *Disparity Computation* StepCapsule instances. Composition over computation space resolves data-dependencies *a priori* and avoids costly runtime book-keeping overhead otherwise required to maintain data-dependencies. However, the majority of the speedup is due to better load balance achieved in coarse-grain composed *Disparity Computation* as described below.

In Figures 48 and 49 the *Queue Imbalance* for the SV application is shown for *Disparity Computation* granularity as coarse (*i.e.,* composed and serialized) and *Disparity Computation* granularity as *fine* (*i.e.,* composed but not serialized), respectively. It is clear from the Queue Imbalance graphs that when serializing the *Disparity Computation*, the imbalance is reduced as discussed earlier.

Figures 42, 43 and 44 illustrate the Total Work done across all PEs during a program execution. The graphs clearly indicate a downward trend in *total work* when increasing the granularity over iteration space for both applications FD and SV. The reduction in total work is again due to a reduction in runtime overhead, as useful work done inside the user-defined stepper function remains constant for all experiments. When granularity is increased over iteration space, there is a total reduction in StepCapsule instances created during the application execution that in turn reduces the total number of synchronization points, distribution and scheduling events and other book-keeping costs. Overall, reducing these events leads to a reduction in total runtime overhead.

Furthermore, percentage overhead in Figures 45, 46 and 47 show that increasing the granularity over iteration space increases the efficiency of the parallel execution by reducing the overhead incurred by the runtime. The same explanation used to describe the Total Work graphs above can be applied here. The percentage overhead metric confirms the hypothesis that composability in Capsules, which even though is more memory intensive due to its dynamic data-structures, can be used to dynamically create efficient coarse-grain computations to reduce the total overhead of parallelization. The fewer coarser-grain computations and fewer synchronization points reduce the ratio of overhead to useful work and improve application's efficient use of the underlying hardware concurrency for speed-up.

The overhead results in Figure 51 for the Synthetic N-Stage pipeline application show that for any number of PEs, increasing the granularity over computation space and serializing the composition helps increase application performance. The reduction in synchronization points and book-keeping cost due to composition of the N-stage pipeline are the main causes of this overhead reduction. The difference in performance is small when the pipeline depth is low (< 10 stages), but performance begins to improve with larger pipeline depths (≥ 10 stages). The improvement in performance is due to the savings achieved from reduced synchronization points and fewer data-dependency book-keeping costs.

Furthermore, the 8 PEs results in Figure 51 show a larger difference in overhead between lower granularity (unserialized composition) and higher granularity (serialized composition) task-graphs than compared to the 1 PE results. The difference in results is due to the higher contention in synchronization points when the application is executing with a higher degree of concurrency (higher number of PEs). Also, there is no significant difference between the Iteration-Major (IM) mode and Computation-Major (CM) mode of serial execution. The similarity in performance is attributed to the lack of composition over iteration-space resulting in equivalent execution schedules for both IM and CM serialization modes.

As for the Apply Filters kernel, we note that changing granularity over iteration space

does not yield improvement in execution time performance as shown in Figure 41. Especially for the 8 PEs experiment, we see performance beginning to degrade with grain sizes larger than 12. The lack of performance improvement is due to the low number of 96 filter instances in the $filterID$ iteration space. The small magnitude of this iteration space does not provide enough opportunity for composition to provide significant reduction in overheads. In fact, composition yields negative performance as it creates a greater queue imbalance in the parallel execution as shown in Figure 50.

Interesting, we note from Figure 52, a comparison of coarse-grain parallel execution using capsules verses fine-grain.parallelism offered by the FFT-based IPP implementation that the coarse-grain Capsules execution scales better than the fine-grain execution of IPP. Since IPP's implementation is a black box, its poor scalability performance is suspected largely to be due to overheads involved in its fine-grain execution. The poor performance of IPP's fine-grain parallel execution clearly shows that too fine a granularity can adversely affect execution performance and scalability.

Overall, composition over computation space and iteration space both help improve parallel execution performance of an application in the absence of abundant hardware parallelism. Composability can be used to dynamically adapt granularity of parallel execution in the application and reduce the parallelization overhead.

CHAPTER X

RELATED WORK

In this chapter we explore related work in the context of parallel programming models and parallelizing compiler based technologies. We begin by exploring the current hardware trend that is becoming pervasive. We then explore some traditional fixed granularity programming models to illustrate the need for parallel programming models to have the semantic ability to support granularity adjustment. We then explore current work attempting to deal with the granularity problem in parallelizing applications. In conclusion we show that Capsules is unique in its ability to provide dynamic granularity control over a unified framework that allows composition over both iteration space and computation space.

10.1 Hardware Evolution Trend

In the past half-decade we have seen the concurrent hardware architecture for shared-memory machines shift from SMP configurations to Multi-Core SMP/CMP to a future where Many-Core chips with over 80 cores are in the horizon [57]. The notion of distributed-memory clusters is slowly fading away into a notion of on-chip distributed-memory architectures such as the Cell B.E processor. Furthermore, these many core architectures are now becoming pervasive at the cost of increasing difficulty for application developers to write programs for such a varying amount of concurrent hardware. Therefore, there are many challenges for designers of parallel programming models and languages to get good performance in such a dynamic and heterogeneous environment. One of such challenges is the granularity problem [5] and in building the Capsules programming model we investigate the challenges in designing such a programming model with general granularity control.

10.2 Parallel Programming Models and Languages

10.2.1 Stampede

Stampede [53, 60, 59] is a parallel programming model for Interactive Streaming time-indexed applications. Stampede supports both distributed and shared memory environments via a built-in inter address space communication library called CLF [54]. Stampede provides a computational model called *spd_threads* that is similar to Pthreads [26]. These *spd_threads* can be dynamically created on any address space and can communicate time-indexed data between each other via abstractions called *channels* and *queues*. The notion of monotonically increasing, single dimensional Virtual Time [30] in Stampede is crucial to its programming model. Virtual Time allows Stampede to provide visibility semantics into the time-indexed data streams for the dynamically instantiated *spd_threads*. The visibility semantics also enables automatic garbage collection [52] and other optimizations such as *Dead Timestamp Identification* [25, 47] and *Adaptive Resource Utilization* [48], within Stampede.

TStreams (Section 10.2.2) and hence Capsules are descendants of the ideas explored in Stampede. The notion of multi-dimensional indexes (Tag instances) used in Capsules is a generalization of the single dimensional monotonically increasing virtual time notion used in Stampede. Similarly, prescribed ItemCapsule Spaces in Capsules are an extension of Stampede's time-indexed data-buffers such as *channels*. One major difference between the two models is in their underlying computational model. For example, computations in Capsules are based on light-weight, functional computation instances, whereas computations in Stampede are heavy-weight, typically long running threads. As mentioned above, *spd_threads* are in fact an extension of Pthreads, whereas Capsule's StepCapsule instances are similar to *Cilk* threads (Section 10.2.7). The common use of the term *thread* in Stampede, Pthreads, and Cilk is often a source of misunderstanding in accurately interpreting the differences between such computational models.

10.2.2 TStreams

Capsules is based on the TStreams [38] programming model and was specifically designed to address the granularity problem in TStreams. As described earlier in Section 2.3, TStreams contains three basic conceptual building blocks used to construct a parallel program. These are: (1) Step Spaces; (2) Item Spaces; and (3) Tag Spaces. These spaces are connected by directed edges to form an application task-graph. These edges or relationships are called: (1) *Producer*; (2) *Consumer* and (3) *Parametrize*. An example of these spaces and their relationships are shown in Figure 1. Note that a TStreams application is not encapsulated in a default StepCapsule Space composition. The notion of composability over computation space is added by Capsules over TStreams.

For each space in a TStreams task-graph, there exist many instance objects during program execution. These instance objects are called Tag, Step and Item instances (Section 2.3). Capsules also expresses the same producer, consumer and parametrize relationships that are used in TStreams. Therefore, the Capsules relationships described in Section 2.3.4 are in fact borrowed from TStreams.

10.2.2.1 Garbage Collection

TStreams allows the user to define a per-Item instance consumer relationship using the *item2consumer()* function. Such consumer relationships are used to enable GC in the runtime by identifying all consumers of a particular Item instance.

When Step instances reach the *executed* state, input Item instances and the parametrizing Tag instance are GC'ed by the TStreams runtime. Item instances are GC'ed when all their consumer Step instances have executed. The GC mechanism requires *either* of the following information to be provided by the programmer when Item instance are produced: (1) The entire set of consuming Step instances to be known (*via* the *item2consumer()* function), or, (2) a reference count for each Item instance be known (*via* the *item2reference()* function).

10.2.2.2 Distribution

TStreams also allows the distribution of Step instances to execute on specified resources using the *stepinstance2resource()* function. The distribution function returns the resource ID when passed a Step instance's parametrizing Tag instance. The same technique is also used within the Capsules runtime for distribution.

10.2.2.3 Optimizations

There are two models of getting a Step's data in TStreams. (1) A Push (Data-flow) model requires the user to provide an *item2consumer()* function and a *consumer2item()* function. The *item2consumer()* function returns the set of Step instances that consume an instance of an Item. The *consumer2item()* function returns a list of Items required to enable a Step instance. The advantage of the Push model is the reduced latency in getting data to its consumer Step instance. (2) A Pull model, on the other hand, requires no *consumer2item()* function, and it is sufficient to enable Step instances when they are prescribed by their Tag instance. The Item instances required for consumption by the Step instance are acquired lazily during the time of the *getItem()* call. The Pull model may incur a higher latency than the Push model but at the benefit to the programmer from having to provide extra book-keeping functions.

10.2.2.4 Limitations of the Model

- **No automatic GC:** Unlike Capsules, GC is not transparent (automatic) in TStreams. GC of Item instances from Item Spaces has to be done via the help of user specified *item2consumer()* or *item2referencecount()* functions. Such a requirement adds additional burden upon the application programmer.

  Furthermore, the TStreams GC mechanism is not adequate for certain dynamic consumer behaviors. Specifically, Items can only be successfully GC'ed if the number of consumers is known at the time of Item production. Applications such as the

Cascade Face Detector where a cascade (list) of classifiers execute based on the output of previous classifiers do not have known consumers until the computation is complete. If any classifier fails to detect a feature, the remaining list of classifier is not executed. Therefore, any input these classifiers may have (such as the image, or the classifier data) would require the undetermined consumption pattern of the classifier stages. The undetermined consumption pattern makes *item2consumer()* and *item2reference()* functions inadequate for GC, which require the consumer Step instances to be known at Item production time.

- **No dynamic granularity adjustment:** The granularity of computations (Steps) or data (Items) cannot be changed in TStreams. Each Step space has fixed code that can run, and there is no mechanism of combining instances from within or across Step spaces. The ratio of runtime system overhead per execution of a Step instance can easily become greater as fine-grain Step spaces are present in the TStreams application task-graph. If Step spaces are fine-grained enough, it may eventually reach a point where the parallel runtime overhead would be too high to attribute any gain in application performance by parallel execution.

10.2.3 Charm++/Charm

Charm++ [32, 35], the successor to Charm [33, 34], is a portable C++ based parallel programming model that uses the object oriented paradigm to represents its core parallel computations. Charm is based on the notion of *Actors* [1], which are concurrent objects that communicate with each other via messages. Actors also allow concurrency within objects but in Charm, the authors consider expressing concurrent objects difficult for application programmers and the runtime to use and maintain. Charm's message driven computational model is essential for latency tolerance. Charm++ uses extensive dynamic load balancing and message prioritization to improve performance.

The computational abstraction used in Charm is the *Chare*, which is old English for

chore. Chares are concurrent tasks that can create new Chares and send messages to other Chares. Chares can also communicate with each other using shared data-structures such as read-only variables, accumulators, monotonic variables and distributed tables. There is also a special Chare called the Branch Office Chare (BOC), an instance of which exists on every processor. BOCs act as replicated objects that normal load-balanced Chares can interact with locally on any given processor.

Charm has been used extensively in real work applications and has a wide user base. However, Charm++/Charm also suffers from the granularity problem and requires the application developer to program Chares with sufficient computation granularity to avoid high parallelization overheads [34]. Their experiments with 3 graph coloring also shows the classic *bath tub* [29] performance result, where increasing granularity of computation improves performance but the graph levels out and then worsens as grain size continues to increase. The authors also state that determining automatic granularity is a hard problem for any given application.

10.2.4 Jade

Jade [43, 69, 70] is a parallel programming *language* designed only for coarse-grain parallelism targeted for heterogeneous platforms. It consists of special language primitives, such as the *withth* representing the paradigm with-and-only-with, that are used to wrap coarse-grain *tasks* along with declarations for shared data access patterns to decompose a serial program into parallelizable computations. In Jade, the read and write accesses are explicitly expressed and are used to resolve dependencies between potentially concurrent tasks.

Jade's programming model also claims to be unsuitable for very fine-grain computations as system overhead would limit the feasibility of parallel execution. Jade's runtime implementation also claims to match exploited concurrency with available concurrency in the parallel machine. However, their approach is fundamentally different from our work in

that the parallel runtime suspends the original task that is responsible for generating new child tasks in order to reduce the load of ready tasks on the parallel machine. Therefore, Jade does not reduce the total number of tasks that execute in the runtime nor does it reduce the synchronization points that add to parallel execution overhead. In our work potential application parallelism is reduced to match hardware parallelism by increasing the grain of fine-grain computations and thereby reducing the number of total computations that interact with the runtime during the lifetime of the entire application.

10.2.5 Linda

Linda [21, 11, 12] is a parallel programming model where distributed client processes interact with each other through read/write calls to a shared abstraction called Tuple Space. The underlying TStreams [38] programming model used to build the Capsules system borrows this concept of Tuples Spaces, calling them Tag Spaces. However, the read semantics in Capsules are different from that in Linda. In Linda, reads from Tuple Spaces is based on pattern matching between a regular expression provided to the read call and the unique Tuples that identify objects in the Tuple Space. Such read semantics can yield multiple objects depending on the number of object-Tuples that match the read regular expression. The read semantics in the Capsules model is intentionally designed to be more strict to disallow non-deterministic regular expression based searches on tag-spaces. The read *get()* calls in Capsules allow queries only by fully defined Tags thereby limiting the *get()* calls to yield only the queried object. The strict read semantics help keep consistency that is required for correctness and a deterministic granularity model.

Another difference between the two models is their underlying execution engine. In Linda, the execution model is primarily process/thread based, where distributed stateful threads constantly loop to perform read/write calls to Tuple spaces. In Capsules, the execution model is based on the higher-level computational abstraction called *StepCapsule*. The scheduling and execution of these StepCapsule instances is supported by an underlying

worker task queue.

Moreover, the higher-level abstraction provided at the programming model level by the Capsules system provides a unifying framework enabling composability for the application developer. Composability in Capsules allow a programmer to adjust the granularity of execution, garbage collection and other features to increase the efficiency of parallel execution based on available hardware concurrency. These abstractions that unify the concept of composability are not provided by the simple abstraction of Tuple spaces in Linda.

10.2.6 Split-C

Split-C [16] is a distributed memory parallel programming extension to the C language developed in the early 1990s. Split-C provides both global and local address spaces along with global and local pointers to allow for efficient distributed parallel processing that can be optimized for locality. Split-C contains special optimizing operators such as the *split-phase assignment* that allows hiding communication latencies by overlapping computation and communication together. Another optimizing operator provided is the *Signaling Store*, which is a form of *push* data-transfer to a remote process. The *Signaling Store* operator is useful for aggressive optimization when the messaging structure of the program is well defined. Distributed data-structures such as *spread arrays* with *spread pointers* to access them have well defined layout to maintain good programmable locality in Split-C programs.

As claimed by its creators, Split-C can in-fact be used to develop higher level parallel programming models. It does not have any notions of composability or granularity control and therefore leaves such responsibilities to the application programmer.

10.2.7 Cilk

Cilk [9] is another high-level programming model that allows the expression of a parallel program as a *spawn tree* of execution, where computational units called *threads* spawn *child* and *successor* threads to enable parallel execution. The *successor* computations are run when the *child* computations complete executing and provide their computed values to

successor via a *continuation* variable. These threads are also non-blocking like *StepCapsules*, and execute to completion once it has been invoked. The Cilk runtime similar to the Capsules runtime is also built around a task queue execution model where each queue maintains the work list for a processor. Cilk uses work-stealing as a primary mechanism to balance the work load among the available queues but also provides optimizations similar to Capsules to distribute StepCapsule instances to the optimal execution queue.

At the programming abstraction level, a Cilk *thread's* execution granularity is fixed. The closest feature in Cilk that resembles composability is over computation space in the *tail call* that runs a new spawned thread immediately after the execution of a parent thread without invoking the scheduler. The compile time *tail call* optimization reduces the cost of parallelization and effectively serializes execution between two computations. In the Capsules model, even though the composition over computation space hierarchy is specified at compile-time, the actual computation StepCapsule instances can dynamically choose at runtime to execute serially in composed coarse-grain form or execute in parallel in its fine-grain computation form.

10.2.8 Intel Thread Building Blocks

In 2007 Intel released the Thread Building Blocks 2.0 [29] (TBB) a high-level parallel programming model for shared-memory Core Multi Processors (CMP). TBB contains the notion of *tasks* representing units of computation rather than the traditional thread model of computation. It allows the programmer to focus on creating task instances and not have to worry about mapping tasks to resources, which is transparently taken care of by its runtime scheduler. TBB is based on Cilk (Section 10.2.7), where tasks can spawn child tasks and either wait for them to return a value, or use a *continuation task* to complete execution.

TBB also provides templates of parallelization such as *parallel_for*, *parallel_reduce* and *parallel_do* for expressing loop parallelism and other templates like *pipeline* and *filters*

to express pipeline and task parallelism. TBB also provides concurrent container data-structures such as the *concurrent_hash_map*, *concurrent_vector* and the *concurrent_queue* to allow shared-memory communication between parallel tasks. There is also a wide variety of synchronization locks available in TBB such the plain *mutex*, *spin_mutex*, *queuing_mutex*, *spin_rw_mutex*, *queuing_rw_mutex* to efficiently synchronize on user-defined critical sections.

One TBB feature relevant to our work is that it enables automatic granularity control. Automatic granularity control is available only for one and two dimensional data-structures, however the model does allow user-defined granularity control for higher dimensional computation and data. Grain size is automatically determined in TBB using the the *auto_partitioner* and the *affinity_partitioner* modules. The difference between them is that the [*affinity_partitioner* also helps keep cache affinity between iteration instances to boost performance. Capsules currently does not support automatic granularity selection and it is left as future work (Section 11.2).

TBB also differs from the Capsules work in that TBB only supports a shared memory model.

Another difference between the two models is that Tasks in TBB can be blocking, whereas StepCapsule instances in Capsules are non-blocking. TBB tasks can be locked on shared data-structures but the synchronized data-access mechanisms in Capsules do not block on unavailable data requests and instead keep track of required data and then restart StepCapsules when the data does become available. The re-execute strategy in the Capsules execution model is doable because of the side-effect free property of StepCapsule instances.

Similar to the tail call mechanism in Cilk described in Section 10.2.7, TBB also has a *scheduler bypass* mechanism that can be used to invoke a child task at the end of a parent task without invoking the scheduler. Such optimizations reduce the scheduling overhead in the runtime. In comparison to mechanisms in Capsules, scheduling overheads are reduced

when coarse-grain computations are formed by composing over computation space and are executed serially using a statically computed non-blocking serialization schedule.

10.2.9 Cell Superscalar (CellSs)

Cell Superscalar (CellSs) [7] is a parallel programming model derived from the Grid Superscalar work [6] for the Cell B.E. [31] architecture. CellSs uses user annotations, similar to those used in OpenMP, to define code blocks eligible for parallel execution on one of Cell's 8 Synergistic Processing Elements (SPEs). The CellSs architecture contains two main components. The first is the source-to-source compiler that transforms the user annotated single source code to a dual source code format, where one source is for the PPE (main thread and management functions) and one on for the SPE (parallel code). The source-to-source compiler also emits DMA transfer calls to the target sources to enable memory access for the SPEs. The second component in CellSs is the CellSs runtime, which manages chores required to enable efficient parallel execution. These chores include resolving data-dependencies of parallel tasks, caching data and code transfer to the local memory of SPEs *etc.* . CellSs also has locality aware scheduling to reduce data-transfers between main-memory and SPE local memory.

CellSs also uses the Octopiler [17] auto-vectorizing compiler to SIMDize the scalar loops inside the annotated SPE code to take advantage of the SIMD data-parallel hardware in the SPEs. Such parallelization strategies are similar to those used by *Ct* (Section 10.4.4), which uses Intel's SSE SIMD instructions to optimize data-parallel codes.

CellSs is different from Capsules in that its superscalar approach allows the application writer to write serial annotated code and parallelism is extracted by analyzing data-dependencies and constructing a task-graph at runtime. Currently Capsules is at a lower level where the user is asked to define a static task-graph to explicitly expose task and data parallelism to the runtime. The data-dependency edge analysis in Capsules is done only once at start-up time, which is sufficient for cycle-free task-graphs allowed in the model.

The static data-dependency analysis differs from the more general data-dependency analysis in CellSs, which is done per task instance but at a cost that adds to the parallel execution overhead of a task.

Finally and most importantly, CellSs does not provide any mechanisms for composability/granularity control. The granularity is fixed, and depends on how much computation the user embeds in the annotated parallelizable code.

10.3 Optimistic Parallelism: Galois

The Galois parallel programming model [41] was designed to extract parallelism from irregular applications that are traditionally difficult to parallelize using static, or even semi-static parallelization techniques due to such application's data-dependent and execution order dependent parallelization possibilities. Even dynamic parallelization techniques such as Thread-Level Speculation (TLS) [74] are too strict and give poor performance for the class of applications targeted by Galois.

Galois provides *optimistic iterator* abstractions such as *Set iterators* and *Ordered-Set iterators* that enables the runtime to know where to optimistically extract parallelism from irregular applications. Furthermore, Galois provides some Object Oriented semantics for creating objects with hooks for the runtime to detect and recover from unsafe accesses to shared-memory.

Although the Galois parallel programming model also dynamically extracts and enables parallelism like Capsules, its general focus is a class of irregular applications not targeted by Capsules. Capsules also enables unbounded data-dependent parallelism just like Galois but the focus of our work again is to enable granularity control to minimize parallelization overhead. Granularity control is not available in Galois and it is targeted for coarse-grain irregular applications.

10.4 Compiler based Parallel Programming Technologies

10.4.1 OpenMP

OpenMP [10] is an emerging standard for parallel programming on shared memory and distributed memory (Cluster OpenMP [42, 27]) architectures that follows the incremental parallelization philosophy. It is a compiler-driven technology, where a serial program is annotated by the user with directives indicating parallelizable regions of code that the compiler can parallelize automatically. For example, in C/C++, the #pragma directives are used to denote parallelizable regions of code. The compiler then adds parallelization code to break-up independent loops iterations into separate executions.

OpenMP also leaves the question of adequate granularity to the application programmer [27]. It recognizes the need to have sufficient granularity in a unit of parallel execution in order to amortize the cost of parallelization. However, the OpenMP standard does not export a programming model level abstraction that enables the programmer to develop the application with the ability to create coarser-grain computations.

10.4.2 Cluster OpenMP

Even though OpenMP started off as targeted for shared memory multiprocessor architectures such as SMPs and CMPs, several efforts have been made to extended the OpenMP 2.0 specification to include a distributed memory architectures [27, 42] such as a Cluster of Workstation (COWs)[1]. It relies on a Software Distributed Shared Memory (SDSM) implementations such as TreadMarks [3] or SCASH [23, 24] to deliver coherent shared memory access to the OpenMP programming model. However, SDSM causes additional overheads such as network traffic and synchronizations to maintain coherence in the abstract shared memory. Moreover, even with the aid of a compiler, it is difficult to optimize parallelization for SDSM as communication for shared-memory is unknown until run time. In fact, context sensitive analysis along with synchronization sensitive analysis has been

[1] A Cluster of Workstations (COWs) is also known as a Network of Workstations (NOWs)

proven to be undecidable [61]. However, efforts have been made to include optimizations to improve the performance of SDSM based OpenMP specification and both experimental and commercial distributed memory OpenMP compilers are available from Omni [42] and Intel [27].

Because of performance problems with SDSM in certain high contention cases, this implementation could yield sub-optimal performance. Nevertheless, with Cluster OpenMP, Intel has shown that the OpenMP model can be extended and made to perform well in distributed memory architectures.

10.4.3 PROMIS parallelizing Compiler

The PROMIS [71] is a multilingual, parallelizing compiler for multiple ISA platforms. Recent work on the PROMIS compiler added support for granularity selection [40] mechanisms targeted for specific hardware. They added Extended Machine Descriptors (EMD) that describe the type and amount of parallelism that can be exploited on a given piece of hardware. EMD can also be used to describe a minimum desired granularity for any hardware resource. These requirements could define the minimum granularity to be in terms of the number loop iterations or number of operations in a piece of functional code. These requirements are then matched when processing the Intermediate Representation (IR) code during the parallelizing phase, which emits OpenMP directives such as *parallel for* to transform the serial code into parallel code that can execute over an OpenMP runtime.

The goals of this work are similar to ours in that they try to reduce the cost of parallelization by avoiding computations that are too fine-grain. However, their approach is fundamentally different in that their granularity adjustment mechanism is based at the level of the auto-parallelizing compiler. Their technique in essence solves the granularity problem in OpenMP by making the automatic parallelization phase of the compiler aware of granularity requirements and emitting OpenMP code that already has acceptable granularity. Such a strategy is akin to a user writing OpenMP code trying to optimize the granularity

by hand, where the user in this case is the automatic parallelizer in PROMIS. Our approach is addressing the granularity issue at the level of the Parallel Programming Model itself, giving the programmer the ability to select granularity in the Capsules framework at runtime.

Another difference is that the approach in PROMIS is static with respect to granularity selection for a target architectures. In PROMIS, specialized code is emitted for a given target architecture, which probably yields very high performance. A different optimized code is emitted for a different target even though the two targets share the same ISA. Our approach is to emit one binary that can be adapted at runtime by profiled parameters that control the granularity of execution dynamically at runtime.

10.4.4 Data-Parallel Models: *Ct*

Ct [22] is a data-parallel programming model that attempts to scale applications without code modification. It attempt to exploit trends seen in future evolution of the ix86 multi/many-core Instruction Architecture (IA). As stated by the *Ct* authors, parallel application performance is dependent on the following factors: 1) core count, 2) vector ISA width, 3) core-to-core latencies, 4) memory hierarchy latencies and 5) synchronization cost. *Ct* targets to abstract out instructions from the binary that are dependent on these factors, and then provide a dynamically linkable solution to optimize the binary to the target ISA. Therefore, the main goal of *Ct* is to provide an intermediary Single Instruction Multiple Data (SIMD) instruction set (known as Virtual Intel Platform (VIP)) to which the data-parallel applications are compiled to. At runtime, *Ct* uses dynamic binding (via the VIP Code Generator (VCG)) to optimize the VIP binary to a particular architecture that uses vector instructions implemented in hardware such as the Intel SSE, SSE2, SSE3 or SSE4 found in successive generations of Intel x86 ISA.

Ct implements a nested data parallel model that is based on works like Nesl [8] and the Paralation Model [19]. The nested data parallel model enables *Ct* to express algorithms and

data-structures commonly difficult to express by other data-parallel programming models including divide-and-conquer algorithms such as sorting, and non-flat data-structures such as sparse matrices and trees. *Ct* includes a basic vector data-type called *TVECs* and vector operators that allow operations on the data.

The data-parallel decomposition focus of *Ct* is clearly at the SIMD ISA level, which is different from the current focus of Capsules at a coarse-grain user-defined data-structure level – An item in Capsules can be anything from a simple integer to a complex graph. Also, the approach in *Ct* is again opposite to that of Capsules wherein *Ct* operations are defined on vector data-structures, which are then decomposed to the appropriate granularity to take advantage of available hardware SIMD concurrency. In Capsules, the user defines finest grain operations, which can be dynamically composed together to form coarse-grain computational and data units.

10.4.5 NVidia CUDA

In 2007, NVidia Corp. released the NVidia CUDA, the Compute Unified Device Architecture programming model. CUDA is targeted to utilize Graphical Processing Units (GPUs) as a data-parallel computing resource. In particular, the NVidia GEForce 8800, 8600, 8500 series, Quadro FX 5600, 4600 series and the Tesla 8 and 10 series chips were targeted for General Purpose (GP) parallel computing.

CUDA is also designed by keeping in mind varying amount of available hardware concurrency. The Tesla architecture, for example, can be combined with multiple GPU boards (such as in the Tesla D870 and S870 configurations) where 2 and 4 Tesla boards respectively could be added to a single computer to enable a (2 x 16) to (4 x 16) multiprocessor [55] configuration. Each multiprocessor is capable of executing 8 threads concurrently (on 8 Scalar Processor (SP) cores) bringing the total number of thread processors from 128 for 1 Tesla configuration to 512 thread processors for a 4 Tesla configuration.

The CUDA programming model supports the basic computation idea of a *thread* representing a function call. The CUDA *thread* is similar to the finest -grain *Step* in TStreams/Capsules. These threads are grouped together in *blocks* of a given size. Multiple blocks comprise of a *grid* and the entire grid represents the data-parallel Kernel application that would execute on the CUDA runtime.

Each thread function in CUDA can be dynamically called by specifying the grid dimension and block dimension. Note that each block executes concurrently on a single multiprocessor independent of other blocks and cannot communicate with them. However, threads within a block can share data. Once thread blocks terminate, new thread blocks are scheduled for execution on a multiprocessor.

In CUDA, *threadIDs* are contiguous N-dimensional spaces (currently limited to only 3 dimensions for convenience). In contrast, Capsules allows the user to reference sparse spaces and allows composing on these sparse iteration spaces according to the programmer's needs.

Furthermore, CUDA breaks thread blocks into *warps* with each warp consisting of 32 parallel threads. Each warp is further divided into a half-warp, which is executed on the NVidia Single Instruction Multiple Thread (SIMT) architecture exposed by the GPUs. SIMT is similar to SIMD, except that it also supports branch divergence between threads in the warp. When threads diverge due to a data-dependent conditional branch, the warp begins to serially execute each branch path taken.

The Capsules work currently does not target SIMD or SIMT architectures and therefore has no optimizations to deal with multiprocessors with such capabilities. However, targeting emerging architectures and developing execution models for them in support of Capsules is part of future research (see Section 11.2).

10.5 Summary

Traditional parallel programming models have lacked the semantic ability to express granularity control and programming in them has required developers to extract only coarse-grain parallelism from applications to minimize overheads. Recently released programming models in past few years have begun to incorporate dynamic granularity control over mainly iteration space, for example, composing computation instances in TBB and CUDA or composing vector data based on vector instruction size in Ct indicate similarity in approach also used in Capsules [49]. However, we enable another dimension of composition over computation space that is not found in other granularity control frameworks to reduce overheads such as synchronization points. Furthermore, the dynamic composition framework in Capsules allows the programmer to control granularity and parallelism at runtime, which is different from other static compiler based granularity control mechanisms such as in PROMIS.

CHAPTER XI

CONCLUSION AND FUTURE WORK

11.1 Conclusions

In this thesis, we have introduced *Capsules* a parallel programming model that brings together two distinct forms of composability. We enumerate composability as *composability over computation space* and *composability over iteration space* and create high-level software abstractions to represent them at the programming model level. The *StepCapsule* abstraction enables composability over Computation Space whereas the *TagCapsule* abstractions enable composability over Iteration Space. Even data can be expressed at varying granularities with the *ItemCapsule* abstraction. We show that this notion of composability is important in enabling the efficient adaptation of a parallel execution to target architectures. We show in our experiments that overheads due to synchronization, maintaining data-dependency and other runtime bookkeeping costs can be minimized by adjusting the granularity of an application's concurrent tasks and moving the synchronization points to the boundary of those coarse-grain computations. Overall, the notion of composability at the programming model level enables the application developers to write parallel applications once, and tune the application granularity parameters later to extract the optimal amount of potential application parallelism required to efficiently utilize the hardware concurrency. We demonstrate the performance benefits of the Capsules Programming Model with both real world and synthetic vision applications and propose future directions for research.

11.2 Future Work

In the Capsules programming model, we provide a uniform framework to dynamically compose and serialize coarse-grain computations over Iteration Space and Computation Space. However, the research done in this dissertation can be taken further to explore other issues such as removing the acyclic task-graph restriction of the model, automatic granularity selection, adding more features to the programming model and supporting and optimizing other hardware platforms.

11.2.1 Cycles in Task-Graphs

As described in Section 3.6 and 4.5, one of the temporary restrictions placed on the Capsule Programming Model was to disallow cycles in the application task-graph. The restriction on cycles was placed to focus our efforts to answer the granularity question and evaluate the performance benefits of our approach. Having achieved the goal of proving performance benefits of dynamic granularity adjustment, it would be an interesting research exercise to evaluate the performance implications of supporting cycles in applications. Allowing cycles to be expressing in Capsules will make the model more general and support a wider class of applications. Applications such as the Fast Fourier Transform (FFT) contain several iterations of an algorithm, which requires cycles to be expressed in the application task-graph.

There are two possible scenarios to be considered when dealing with cycles in the Capsules task-graph. The first is cycles within compositions over computation space and the second is cycles formed across compositions over computation space. Careful consideration to possible task-graph cases would have to be given along with adding to rules to disallow potential problem cases.

Generally, the acyclic graph restriction could be removed by allowing a stepper function to re-define the matching dimensions between the computation and the data's iteration space at certain edges possibly marked as cyclic. User-defined Mapping functions at such

edges could then be used by the runtime for each instance of a stepper to determine a shift in the iteration dimension values. However, it is important that mapping remain consistent, which can be either checked by the runtime at an added overhead cost, or be left at the discretion of the application developer. Also, required would be to remove checks in the task-graph analysis phase that currently disallow cycles. However, it would still be important to detect cycles so that the serialization schedule for composition over computation space would know whether to re-execute a schedule as the cycle could have potentially generated new computations via emitting new TagCapsule instances.

11.2.2 Auto-tuning Framework

The optimal granularity of execution is difficult to determine without knowing the target hardware platform and understanding the affects of adjusting the application's computation and data granularity. Moreover, the large space of valid argument values for tunable parameters is difficult to explore manually. Therefore, there is a need to automatically and efficiently find optimal granularity values so as to remove this burden from application developers.

Tunable parameters for a parallel program are the different iteration space dimensions of a computation, and computation space code pieces where hierarchical composition can occur to adjust granularity. Furthermore, when having dual compositions over both iteration space and computation space, different serialization schedules are also possible, namely Iteration-Major and Computation-Major serialization that maintain locality in different ways. Given a hardware platform, these auto-tuning tools could efficiently search the space of tunable parameters for an optimal configuration for the application's execution. The auto-tuning framework (*auto-tuners*) should automatically probe these spaces and find an optimal configuration that increases application execution. Auto-tuners have also been proposed by earlier work [5] as a possible solution to automatically determine granularity.

It can be noted from Sections 4.7 and 4.9 that granularity over iteration space is defined by the user at two sets of statically known points in a Capsules task-graph, namely at Dimensional Expansion points and at Dimensional Reduction points. Any dynamic auto-tuning framework that would attempt to adjust the granularity would have to be aware of such points and keep consistency between the granularity chosen for iteration dimension instances that are common between such points. For example, assume a function *foo()* iterating over the dimension $< i >$ is expanding and defining the dimension j into another iteration space $< i, j >$. Also, a function *bar()* is iterating over dimension $< i >$ and performing a dimensional reduction by consuming from data parametrized by an iteration space $< i, j >$. In this case, both edges namely the one from *foo()* and the one into *bar()* much keep consistency between the granularity chosen for instances of dimension j.

The general steps used by the Auto-tuning framework is described below:

1. Profile the parallel execution to measure overheads and performance.
2. Adjust the granularity of execution to reduce these overheads.
3. Adjust the distribution of tasks to improve locality.
4. Repeat profiling process above until no further change in execution speed-up is observed.

11.2.3 Metrics Feedback API

One of the requirements for an auto-tuning framework for granularity selection would be a profiling or monitoring framework to measure the performance of the application or runtime. A dynamic feedback mechanism would therefore be required that can provide hints on various performance metrics about the application, runtime or both. Metrics such as those described in Section 9.1.1 could be made available to such a granularity selection framework via a generic API interface. A light-weight back-end data capture support would be required that could collect metric data over certain time intervals or even by executing

the application kernels and then performing some post-mortem analysis accessible by the API.

11.2.4 Finer-grain Analysis of Runtime Overheads

As described earlier, there are four main causes of overhead in a parallel runtime:

1. Dynamic data-structure cost or Book-keeping cost.
2. Distribution cost.
3. Scheduling cost.
4. Synchronization cost or Data-access cost

Although the percentage overhead metric (see Section 9.1.1) provided in this dissertation coalesces all the above mentioned overheads, it would be an interesting study to investigate the exact cost breakdown for each *fine-grain* cause of overhead listed above and other such as GC overhead. In order to produce such *fine-grain* metrics using OProfile [46], the samples between all different code paths in the runtime would have to be separated and summed individually. Perhaps the code annotation feature *opannotate* available in OProfile along with some custom scripts to separate and sum up profiling information could be used to extract finer-grain metrics.

We had also attempted to retrieve fine-grain information on synchronization cost by insert timing calls and logging custom events using the framework used to extract the Queue Imbalance metric. However, the high frequency of synchronization especially when executing in fine-grain created high overheads in the timing infrastructure and skewed results with respect to coarse-grain execution. The OProfile infrastructure, on the other hand, uniformly samples all code doesn't suffer from result skew when certain code paths are frequency called.

11.2.5 Check-point Restart

One of the benefits of the Capsules framework is that it enables check-point-restart or migration of computations. These features allow reducing the cost of failures in large parallel architectures where failures are common and restarting a computation from the beginning would otherwise be extremely costly. The Capsules framework enables check-pointing at different granularities, with coarse-grain check-pointing enabling recovery at the same grain. The auto-tuning tools will help determine the correct granularity for check-pointing such that application performance is unaffected.

11.2.6 Distributed Memory Architectures

It would also be interesting to extend the current shared-memory implementation of the Capsules Programming Model to support a distributed memory environment. We are currently designing and implementing a runtime using MPI [50, 80, 79], although many optimizations would have to be made before any interesting results could be presented. The Cell B.E. [31] processor would also make an interesting test-bed to evaluate a distributed memory implementation. Each Cell B.E. processor contains a Power Processing Elements (PPE) and 8 Synergistic Processing Elements (SPEs), where each SPE has its own 256kB of *local store* memory that is not coherent with the on-die cache that is accessible to the PPE. Memory accesses to main-memory have to be explicitly scheduled via DMA to the local store, which makes the Cell B.E. an interesting distributed memory architecture to explore with the Capsules Programming Model.

11.2.7 Distribution, Scheduling, Work Stealing and Granularity Control

Within a distributed memory architecture, there are a host of interesting questions that naturally arise concerning the effects of distribution, scheduling and work stealing especially when dealing with granularity control. It is without a doubt that granularity control adds another dimension of complexity to the already difficult problem of resource management

in distributed environments. Such compounding issues create a strong case to investigate how automatic granularity selection can be enabled for a given parallel program.

11.2.8 Dynamic Resource Availability

In the performance analysis of Capsules (Chapter 9) we keep the number of resources constant for the duration of application execution for each of the application test cases. Therefore, we have not considered true dynamic change of resources in our experiments. However, changing resources also means having the ability to adapt scheduling and distribution of tasks. Some of these issues can be dealt with dynamic adaptation techniques such as Work Stealing, but scheduling policies can be a major issue dictating performance. Changing granularity during program execution of course, would also play a role in application performance. These issues are not exhaustively considered in our work and would be an interesting topic for future research.

11.2.9 Design Patterns and Code Re-use

The clean ability in the underlying TStreams model to express maximal potential parallelism and Capsule's ability to express dynamically adjustable granularity could be exploited as a fundamental property to help build a class of re-usable generic parallel design patterns. Here certain Capsules task-graphs could be re-used for many different applications. For example, the Apply Filters kernel task-graph can theoretically be re-used by replacing the Apply Filter functions depending on the application.

Furthermore, the notion of composability also enables code re-use, where Capsule applications could be combined with other Capsules applications to form further complicated task-graphs. To achieve such inter task-graph merging and composability, more work would be needed to explicitly define interfaces that specify interactions of stepper functions with the environment. Also required would be the ability to expand the iteration dimensions of a task-graph such that it can be called multiple times even when combined with other task-graphs.

APPENDIX A

EDGE DATA-STRUCTURES

The data-structures *OutTagEdge_t* and *OutItemEdge_t* are the same except for the referenced Space type. Due to their similarities, only *OutItemEdge_t* is illustrated below.

```
class InItemEdge_t {
  ItemCapsuleSpace_t *space;
  GrainDefinitions_t  s2ifuncs;
  int                 edgeIndex;
  int                 startDim;
  int                 numMatchingAxes;
  bool                external;
  int                 externalIndex;
};
```

```
class OutItemEdge_t {
  ItemCapsuleSpace_t *space;
  int                 edgeIndex;
  int                 numMatchingAxes;
  bool                external;
  int                 externalIndex;
};
```

APPENDIX B

CODE: SERIAL EXECUTION, ITERATION-MAJOR

Code for the following functions is included below:

- *autoSCFuncWithoutGet_IM()*
- *autoSCFuncWithGetFSE_IM()*

```
1  bool StepCapsule_t::autoSCFuncWithoutGet_IM
     (const DataTree_t&    dt,
3     const TagCapsule_t& tc,
      const int           dim,
5     Tag_t               iTag,
      Tag_t               rTag)
7  {
    int numAxes = space->getNumAxes();
9   if(dim == 0) {
      // get the NullTagCapsule dependent data from
11    // NullTagCapsuleSpace parametrized ItemCapsules
      assignItemCapsulesFromDataTree(dt, 0 /* dim */);
13    assignItemFromItemCapsule(0 /* dim */, iTag, rTag);
      if(numAxes == 0) { // special case
15      // setup environment before unfolding last dim Int-Range Node
        initOutDataStructures();
17      StepCapsuleFunction_t func = space->getFunc();
        func(env, iTag); // execute!
19    } else {// numAxes > 0
        bool retVal = autoSCFuncWithoutGet_IM(dt,
21        tc, 1 /* dim + 1 */, iTag, rTag);
      }// if-else {}
```

```
23  } else { // if(dim >= 1)
      // Now, iterate through the next dimension via
25    // the child TagCapsules to execute or recurse
      TagCapsule_t::const_iterator iter = tc.begin();
27    for(int d = 0; iter != tc.end(); ++iter, d++) {
        const TagCapsule_t& ctc = *iter;
29      const IntRange_t& cValue = ctc.getValue();
        const DataTree_t& cdt = *(dt.children[d]);
31      // Assign ItemCapsule from iterator; Increment iterator
        assignItemCapsulesFromDataTree(cdt, dim);
33      if(dim == numAxes) {
          // setup environment before unfolding last dim IntRange Node
35        initOutDataStructures();
        }
37      // append int values @ location 'dim' to rTag and iTag
        rTag.modifyValue(dim - 1, cValue);
39      int lower = cValue.l,
          upper = ((cValue.isRange == true)?cValue.u:cValue.l);
41      for(int i = lower; i <= upper; i++) {
          iTag.modifyValue(dim - 1, i);
43        // save Items from ItemCapsule to ItemHolder;
          assignItemFromItemCapsule(dim, iTag, rTag);
45        if(dim == numAxes) {
            StepCapsuleFunction_t func = space->getFunc();
47          func(env, iTag); // execute!!
          } else {
49          // autoSCFuncWithoutGet_IM(dim + 1);
            bool retVal = autoSCFuncWithoutGet_IM(cdt,
51            ctc, dim + 1, iTag, rTag);
          }
53      }// for(lower--upper)
        // increment leafNodeIndex when unfolding last dim IntRange Node
```

```
55        if(dim == numAxes) {
            env.leafNodeIndex++;
57        }
      }// for(iter)
59    }// if-else block()
    return true;
61  }// autoSCFuncWithoutGet_IM()
```

```
1   bool StepCapsule_t::autoSCFuncWithGetFSE_IM
      (DataTree_t&          dt,
3      const TagCapsule_t& tc,
       const int           dim,
5      Tag_t               iTag,
       Tag_t               rTag,
7      bool                blocked)
    {
9     int numAxes = space->getNumAxes();
      if(dim == 0) {
11      bool getRetVal = getFullSpecifiedEdgeData(&dt,
          0, TagCapsule_t::NullTagCapsule);
13      if(getRetVal == false) { blocked = true; }
        // get the NullTagCapsule dependent data
15      // from NullTagCapsuleSpace parametrized ItemCapsules
        if(blocked == false) {
17        assignItemCapsulesFromDataTree(dt, 0 /* dim */);
          assignItemFromItemCapsule(0 /* dim */, iTag, rTag);
19      }
        if(numAxes == 0) { // for special cases when (numAxes == 0)
21        if(blocked == false) {
            // setup environment before unfolding last dim IntRange Node
23          initOutDataStructures();
            StepCapsuleFunction_t func = space->getFunc();
25          func(env, iTag); // execute!
```

```
      }// if(blocked == true)
    } else { // (numAxes > 0)
      bool retVal = autoSCFuncWithGetFSE_IM(dt,
        tc, 1 /* dim + 1 */, iTag, rTag, blocked);
      if(retVal == false) { blocked = true; }
    }// end of if-else block
  } else { // if(dim >= 1)
   // get data for dim and initialize data iterators
    bool getRetVal = getFullSpecifiedEdgeData(&dt, dim, tc);
    if(getRetVal == false) { blocked = true; }
    // Now, iterate through the next dimension via
    // the child TagCapsules to execute or recurse
    TagCapsule_t::const_iterator iter = tc.begin();
    for(int d = 0; iter != tc.end(); ++iter, d++) {
      const TagCapsule_t& ctc = *iter;
      const IntRange_t& cValue = ctc.getValue();
      // create a new child data-tree node
      DataTree_t* cdt = new DataTree_t(cValue);
      dt.children.push_back(cdt);
      if(blocked == false) {
        // assign child ItemCapsule nodes from dim-1 to dim
        assignChildICtoChildDT(*cdt, dt, d, dim, true);
        // Assign ItemCapsule from iterator; Increment iterator
        assignItemCapsulesFromDataTree(*cdt, dim);
      }// if(blocked == false)
      // append int values @ location 'dim' to rTag and iTag
      rTag.modifyValue(dim - 1, cValue);
      if(blocked == false && dim == numAxes) {
        // setup environment before unfolding last dim IntRange Node
        initOutDataStructures();
      }
      int lower = cValue.l,
```

```
        upper = ((cValue.isRange == true)?cValue.u:cValue.l);
59      for(int i = lower; i <= upper; i++) {
          iTag.modifyValue(dim - 1, i);
61        if(blocked == false) {
            // save Items from ItemCapsule to ItemHolder;
63          assignItemFromItemCapsule(dim, iTag, rTag);
          }
65        if(dim == numAxes) {
            if(blocked == false) {
67            StepCapsuleFunction_t func = space->getFunc();
              func(env, iTag); // execute!!
69          }// if(blocked == false)
          } else {
71          // autoSCFuncWithGetFSE_IM(dim + 1);
            bool retVal = autoSCFuncWithGetFSE_IM(*cdt,
73              ctc, dim + 1, iTag, rTag, blocked);
            if(retVal == false) { blocked = true; }
75        }
        }// for(lower--upper)
77      // increment leafNodeIndex when unfolding
        // last dim IntRange Node
79      if(blocked == false && dim == numAxes) {
          env.leafNodeIndex++;
81      }
      }// for(iter)
83  }// if-else block()
    return (!blocked);
85 }// autoSCFuncWithGetFSE_IM()
```

APPENDIX C

CODE: SERIAL EXECUTION, COMPUTATION-MAJOR

Code for the following functions is included below:

- *autoSCFuncWithGetFSEandPSE_CM()*[1]
- *executeCG_SerialSchedule_CM()*

```
1  bool StepCapsule_t::autoSCFuncWithGetFSEandPSE_CM
     (DataTree_t&          dt,
3     const TagCapsule_t& tc,
      const int           dim,
5     Tag_t               iTag,
      Tag_t               rTag,
7     bool                blocked)
   {
9    int numAxes = space->getNumAxes();
     if(dim == 0) {
11     bool getRetVal = getFullSpecifiedEdgeData(&dt,
         0, TagCapsule_t::NullTagCapsule);
13     if(getRetVal == false) { blocked = true; }
       // get the NullTagCapsule dependent data
15     // from NullTagCapsuleSpace parametrized ItemCapsules
       if(blocked == false) {
17       assignItemCapsulesFromDataTree(dt, 0 /* dim */);
         assignItemFromItemCapsule(0 /* dim */, iTag, rTag);
19     }
       if(numAxes == 0) { // special case
```

[1]The illustrated code for *autoSCFuncWithGetFSEandPSE_CM()* only gets data over FSIEs and not PSIEs. Getting PSIEs during CM execution-mode has not been implemented

```
21        if(blocked == false) {
            // setup environment before unfolding last dim IntRange Node
23          initOutDataStructures();
            bool retVal = executeCG_SerialSchedule_CM(iTag); // execute!!
25        }// if(blocked == true)
        } else {
27        bool retVal = autoSCFuncWithGetFSEandPSE_CM(dt,
             tc, 1 /* dim + 1 */, iTag, rTag, blocked);
29        if(retVal == false) { blocked = true; }
        }// end of if-else block
31    } else { // if(dim >= 1)
       // get data for dim and initialize data iterators
33      bool getRetVal = getFullSpecifiedEdgeData(&dt, dim, tc);
        if(getRetVal == false) { blocked = true; }
35      // Now, iterate through the next dimension via
        // the child TagCapsules to execute or recurse
37      TagCapsule_t::const_iterator iter = tc.begin();
        for(int d = 0; iter != tc.end(); ++iter, d++) {
39        const TagCapsule_t& ctc = *iter;
          const IntRange_t& cValue = ctc.getValue();
41        // create a new child data-tree node
          DataTree_t* cdt = new DataTree_t(cValue);
43        dt.children.push_back(cdt);
          if(blocked == false) {
45          // assign child ItemCapsule nodes from dim-1 to dim
            assignChildICtoChildDT(*cdt, dt, d, dim, true);
47          // Assign ItemCapsule from iterator; Increment iterator
            assignItemCapsulesFromDataTree(*cdt, dim);
49        }// if(blocked == false)
          if(blocked == false && dim == numAxes) {
51          // setup environment before unfolding last dim IntRange Node
            initOutDataStructures();
```

```
53        }
          // append int values @ location 'dim' to rTag and iTag
55        rTag.modifyValue(dim - 1, cValue);
          int lower = cValue.l,
57            upper = ((cValue.isRange == true)?cValue.u:cValue.l);
          for(int i = lower; i <= upper; i++) {
59          iTag.modifyValue(dim - 1, i);
            if(blocked == false) {
61            // save Items from ItemCapsule to ItemHolder;
              assignItemFromItemCapsule(dim, iTag, rTag);
63          }
            if(dim == numAxes) {
65            if(blocked == false) {
                bool retVal = executeCG_SerialSchedule_CM(iTag); // execute!!
67            }// if(blocked == false)
            } else {
69            // autoSCFuncWithGetFSEandPSE_CM(dim + 1);
              bool retVal = autoSCFuncWithGetFSEandPSE_CM(*cdt,
71                ctc, dim + 1, iTag, rTag, blocked);
              if(retVal == false) { blocked = true; }
73          }
          }// for(lower--upper)
75        // increment leafNodeIndex when unfolding last dim IntRange Node
          if(blocked == false && dim == numAxes) {
77          env.leafNodeIndex++;
          }
79      }// for(iter)
      }// if-else block()
81    return (!blocked);
    }// autoSCFuncWithGetFSEandPSE_CM()
```

```
    bool StepCapsule_t::executeCG_SerialSchedule_CM(const Tag_t& tag) {
2     const StepCapsuleSpace_t::SCSVec_t& schedule = space->getSchedule();
```

```
#if SANITY_CHECK
  // Need to prescribe inner ItemCapsules
  const StepCapsuleSpace_t::ItemCapsuleSpaces_t& childISpaces
    = space->getChildItemCapsuleSpaces();
  StepCapsuleSpace_t::ItemCapsuleSpaces_t::const_iterator iIter
    = childISpaces.begin();
  for(; iIter != childISpaces.end(); ++iIter) {
    const string& sName = iIter->first;
    const ItemCapsuleSpace_t *iSpace = iIter->second;
    ItemCapsulePool_t* iPool = icPools[sName];
    if(iSpace->getParentPrescribed()) {
      iPool->addPrescribed(TagCapsule_t::makeTagCapsule(tag));
    }
  }// for(iIter)
#endif // SANITY_CHECK
  for(int i = resumeIndex; i < static_cast<int>(schedule.size()); i++) {
    StepCapsuleSpace_t* scs = schedule[i];
    StepCapsulePool_t* scsPool = scPools[scs->getName()];
    // When executing in computation-major mode, we have to add the
    // prescribed SC into the pool because it was not added earlier.
    if(scsPool->getSpace()->getParentPrescribed()) {
      scsPool->addPrescribed(TagCapsule_t::makeTagCapsule(tag));
    }
    bool retVal = scsPool->run();
    if(retVal == false) {
      resumeIndex = i;
      break;
    }
  }// for(len)
  return retVal;
}// executeCG_SerialSchedule_CM()
```

REFERENCES

[1] AGHA, G., *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.

[2] AMDAHL, G. M., "Validity of the single-processor approach to achieving large scale computing capabilities," in *AFIPS Conference*, vol. 30, (Atlantic City, NI), pp. 483–485, AFIPS Press, Reston, VA, April 18-20 1967.

[3] AMZA, C., COX, A. L., DWARKADAS, S., and KELEHER, P., "TreadMarks: Shared Memory Computing on Networks of Workstations," *IEEE Computer*, vol. 29, pp. 18–28, February 1996.

[4] ARVIND and NIKHIL, R. S., "Executing a program on the MIT tagged-token dataflow architecture," *IEEE Transactions on Computers*, vol. 39, pp. 300–318, March 1990.

[5] ASANOVIC, K., BODIK, R., CATANZARO, B. C., GEBIS, J. J., HUSBANDS, P., KEUTZER, K., PATTERSON, D. A., PLISHKER, W. L., SHALF, J., WILLIAMS, S. W., and YELICK, K. A., "The Landscape of Parallel Computing Research: A View from Berkeley," Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[6] BADIA, R., LABARTA, J., SIRVENT, R., PÉREZ, J., CELA, J., and GRIMA, R., "Programming Grid Applications with GRID Superscalar," *Journal of Grid Computing*, vol. 1, no. 2, pp. 151–170, 2003.

[7] BELLENS, P., PEREZ, J., BADIA, R., and LABARTA, J., "CellSs: A programming model for the Cell BE architecture," *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[8] BLELLOCH, G., "NESL: A nested data-parallel language," Tech. Rep. CMU-CS-93-129, School of ComputerScience, Carnegie-Mellon University, April 1993.

[9] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., and ZHOU, Y., "Cilk: An Efficient Multithreaded Runtime System," in *PPOPP '95: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (New York, NY, USA), pp. 207–216, ACM Press, 1995.

[10] BOARD, O. A. R., "OpenMP: Simple, Portable, Scalable SMP Programming." http://www.openmp.org, Accessed: June, 2008.

[11] CARRIERO, N. and GELERNTER, D., "Linda in Context," *Commun. ACM*, vol. 32, no. 4, pp. 444–458, 1989.

[12] CARRIERO, N. and GELERNTER, D., "A Computational Model of Everything," *Communications of the ACM*, vol. 44, pp. 77–81, November 2001.

[13] CARTER, L., FERRANTE, J., HUMMEL, S. F., ALPERN, B., and GATLIN, K.-S., "Hierarchical Tiling: A Methodology for High Performance," Tech. Rep. CS-96-508, University of California at San Diego, San Diego, CA, 1996.

[14] CHRISTIAN, A. D. and AVERY, B. L., "Digital Smart Kiosk Project," in *ACM SIGCHI '98*, (Los Angeles, CA), pp. 155–162, April 18–23 1998.

[15] CMU/VASC IMAGE DATABASE, "The Combined MIT/CMU Test Set with Ground Truth for Frontal Face Detection." http://vasc.ri.cmu.edu/idb/html/face/frontal_images/index.html, Accessed: June, 2008.

[16] CULLER, D. E., DUSSEAU, A., GOLDSTEIN, S. C., KRISHNAMURTHY, A., LUMETTA, S., VON EICKEN, T., and YELICK, K., "Parallel programming in Split-C," *Supercomputing'93. Proceedings*, pp. 262–273, 1993.

[17] EICHENBERGER, A., O'BRIEN, J., O'BRIEN, K., WU, P., CHEN, T., ODEN, P., PRENER, D., SHEPHERD, J., SO, B., SURA, Z., and OTHERS, "Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture," *IBM Systems Journal*, vol. 45, no. 1, pp. 59–84, 2006.

[18] GA TECH, ROBOTICS & INTELLIGENT MACHINES, "Learning Applied to Ground Robots (LAGR) Project." Student: Dongshin Kim; Principal Investigators: Tucker Balch, Aaron Bobick, Frank Dellaert, Magnus Egerstedt and James M. Rehg, http://borg.cc.gatech.edu/lagr, Accessed: June, 2008.

[19] GARY W. SABOT, *The Paralation Model: Architecture-Independent Parallel Programming*. Cambridge, MA, USA: MIT Press, September 1988.

[20] GEHANI, N., "Capsules: A shared memory access mechanism for concurrent c/c++," *IEEE Transactions on Parallel and Distributed Systems*, vol. 04, no. 7, pp. 795–811, 1993.

[21] GELERNTER, D., "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, 1985.

[22] GHULOUM, A., SMITH, T., WU, G., ZHOU, X., FANG, J., GUO, P., SO, B., RAJAGOPALAN, M., CHEN, Y., and CHEN, B., "Future-Proof Data Parallel Algorithms and Software on Intel Multi-Core Architecture," *Intel Technology Journal*, vol. 11, pp. 333–347, November 2007.

[23] HARADA, H., TEZUKA, H., HORI, A., SUMIMOTO, S., TAKAHASHI, T., and ISHIKAWA, Y., "Implementation and Evaluation of Memory Barrier on Software Distributed Shared Memory on Myrinet," in *JSPP'99*, pp. 237–244, June 1999.

[24] HARADA, H., TEZUKA, H., HORI, A., SUMIMOTO, S., TAKAHASHI, T., and ISHIKAWA, Y., "SCASH: Software DSM using High performance network on commodity hardware and software," in *In Eight Workshop on Scalable Shared-Memory Multiprocessors*, pp. 26–27, ACM, May 1999.

[25] HAREL, N., MANDVIWALA, H. A., KNOBE, K., and RAMACHANDRAN, U., "Dead Timestamp Identification in Stampede," in *The 2002 International Conference on Parallel Processing (ICPP'02)*, (Vancouver, BC, Canada), August 2002.

[26] IEEE, "Threads standard POSIX 1003.1c-1995 (also ISO/IEC 9945-1:1996)," 1996.

[27] INTEL, "C++ Compiler 10.1 for Linux." With OpenMP support, http://www.intel.com/cd/software/products/asmo-na/eng/compilers/277618.htm, Accessed: June, 2008.

[28] INTEL, "Core 2 Quad Processors." http://www.intel.com/products/processor/core2quad, Accessed: June, 2008.

[29] INTEL, "Thread Building Blocks 2.1." http://www.threadingbuildingblocks.org, Accessed: June, 2008.

[30] JEFFERSON, D. R., "Virtual Time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 404–425, July 1985.

[31] KAHLE, J. A., DAY, M. N., HOFSTEE, H. P., JOHNS, C. R., MAEURER, T. T., and SHIPPY, D., "Introduction to the Cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, pp. 589–604, July/August 2005.

[32] KALE, L. V. and KRISHNAN, S., "CHARM++: A Portable Concurrent Object Oriented System Based on C++," *SIGPLAN Notices*, vol. 28, no. 10, pp. 91–108, 1993.

[33] KALE, L. V., RAMKUMAR, B., SINHA, A., and GURSOY, A., "The Charm Parallel Programming Language and System: Part I - Description of Language Features," Tech. Rep. 95-02, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[34] KALE, L. V., RAMKUMAR, B., SINHA, A., and GURSOY, A., "The Charm Parallel Programming Language and System: Part II - The Runtime System," Tech. Rep. 95-02, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[35] KALE, L. and KRISHNAN, S., "Charm++: Parallel Programming with Message-Driven Objects," *Parallel Programming using C+*, pp. 175–213, 1996.

[36] KANG, S. B., "A Survey of Image-based Rendering Techniques," Tech. Rep. CRL 97/4, Digital Equipment Corp., Cambridge Research Lab., August 1997.

[37] KANG, S. B., "Image-Based Rendering," tech. rep., Digital Equipment Corporation, Cambridge Research Laboratory, 1998.

[38] KNOBE, K. and OFFNER, K., "TStreams: How to Write a Parallel Program," Tech. Rep. HPL-2004-193, Hewlet Packard Labs - Cambridge Research Laboratory, Cambridge, MA, 2004.

[39] KNOBE, K., REHG, J. M., CHAUHAN, A., NIKHIL, R. S., and RAMACHANDRAN, U., "Scheduling Constrained Dynamic Applications on Clusters," in *Proc. SC99: High Performance Networking and Computing Conference*, (Portland, OR), November 1999.

[40] KO, W. and POLYCHRONOPOULOS, C. D., "Automatic Granularity Selection and OpenMP Directive Generation via Extended Machine Descriptors in the PROMIS Parallelizing Compiler," in *2nd International Workshop on OpenMP (IWOMP 2006)*, (Reims, France), June 12–15 2006.

[41] KULKARNI, M., PINGALI, K., WALTER, B., RAMANARAYANAN, G., BALA, K., and CHEW, L., "Optimistic Parallelism Requires Abstractions," *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pp. 211–222, 2007.

[42] KUSANO, K., SATOH, S., and SATO, M., "Performance Evaluation of the Omni OpenMP Compiler," in *Proc. for Third International Symposium on High Performance Computing, (ISHPC), Springer-Verlag LNCS 892*, (Tokyo, Japan), p. 403, October 16–18 2000.

[43] LAM, M. S. and RINARD, M. C., "Coarse-grain parallel programming in Jade," in *PPOPP '91: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, (New York, NY, USA), pp. 94–105, ACM Press, 1991.

[44] LEE, E. A., "The Problem with Threads," Tech. Rep. UCB/EECS-2006-1, EECS Department, University of California at Berkley, 2006.

[45] LEHMAN, T. J., MCLAUGHRY, S. W., and WYCKOFF, P., "TSpaces: The Next Wave," in *Hawaii International Conference on System Sciences (HICSS-32)*, January 1999.

[46] LEVON, J., "OProfile, A System-wide profiler for Linux systems." http://oprofile.sourceforge.net, Accessed: June, 2008.

[47] MANDVIWALA, H. A., HAREL, N., KNOBE, K., and RAMACHANDRAN, U., "A Comparative Study of Stampede Garbage Collection Algorithms," in *The 15th Workshop on Languages and Compilers for Parallel Computing*, (College Park, MD), July 2002.

[48] MANDVIWALA, H. A., HAREL, N., RAMACHANDRAN, U., and KNOBE, K., "Adaptive resource utilization via feedback control for streaming applications," in *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, (Washington, DC, USA), p. 69.1, IEEE Computer Society, 2005.

[49] MANDVIWALA, H. A., RAMACHANDRAN, U., and KNOBE, K., "Capsules: Expressing composable compositions in a parallel programming model," in *The 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC 07)*, (University of Illinois at Urbana-Champaign, Urbana, Illinois), October 11-13 2007.

[50] MARC SNIR AND STEVE OTTO AND STEVEN HUSS-LEDERMAN AND DAVID WALKER AND JACK DONGARRA, *MPI - The Complete Reference. Volume 1 - The MPI Core. Second Edition*, vol. 1. Cambridge, MA, USA: MIT Press, September 1998.

[51] NEILL, D. and WEIRMAN, A., "On the Benefits of Work Stealing in Shared-Memory Multiprocessors," tech. rep., Department of Computer Science, Carnegie Mellon University, 2006.

[52] NIKHIL, R. S. and RAMACHANDRAN, U., "Garbage Collection of Timestamped Data in Stampede," in *Proc. Nineteenth Annual Symposium on Principles of Distributed Computing (PODC 2000)*, (Portlan, Oregon), July 2000.

[53] NIKHIL, R. S., RAMACHANDRAN, U., REHG, J. M., HALSTEAD, JR., R. H., JOERG, C. F., and KONTOTHANASSIS, L., "Stampede: A programming system for emerging scalable interactive multimedia applications," in *Proc. Eleventh Intl. Wkshp. on Languages and Compilers for Parallel Computing (LCPC 98)*, (Chapel Hill, NC), August 7-9 1998.

[54] NIKHIL, R. S. and PANARITI, D., "*CLF*: A common Cluster Language Framework for Parallel Cluster-based Programming Languages," tech. rep., Digital Equipment Corporation, Cambridge Research Laboratory, 1998.

[55] NVIDIA CORP., "NVIDIA CUDA Compute Unified Device Architecture, Programming Guide." Version 2.0 Beta2, http://www.nvidia.com/object/cuda_develop.html, Accessed: June, 2008.

[56] OFFNER, C. and KNOBE, K., "Weak Dynamic Single Assignment Form," Tech. Rep. HPL-2003-169R1, Hewlet Packard Labs - Cambridge Research Laboratory, Cambridge, MA, 2003.

[57] OTELLINI, P., "Intel Developer Forum (IDF)," 2006.

[58] RAMACHANDRAN, U., KNOBE, K., HAREL, N., and MANDVIWALA, H. A., "Distributed Garbage Collection Algorithms for Timestamped Data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 10, pp. 1057–1071, 2006.

[59] RAMACHANDRAN, U., NIKHIL, R., REHG, J. M., ANGELOV, Y., ADHIKARI, S., MACKENZIE, K., HAREL, N., and KNOBE, K., "Stampede: A Cluster Programming Middleware for Interactive Stream-oriented Applications," *IEEE Transactions on Parallel and Distributed Systems*, 2003.

[60] RAMACHANDRAN, U., NIKHIL, R. S., HAREL, N., REHG, J. M., and KNOBE, K., "Space-Time Memory: A Parallel Programming Abstraction for Interactive Multimedia Applications," in *Proc. Principles and Practice of Parallel Programming (PPoPP'99)*, (Atlanta, GA), May 1999.

[61] RAMALINGAM, G., "Context-sensitive synchronization-sensitive analysis is undecidable," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 22, no. 2, pp. 416–430, 2000.

[62] RAMANUJAM, J. and SADAYAPPAN, P., "Tiling Multidimensional Iteration Spaces for Nonshared Memory Machines," *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pp. 111–120, 1991.

[63] RAMANUJAM, J. and SADAYAPPAN, P., "Tiling Multidimensional Iteration Spaces for Multicomputers," *Journal of Parallel and Distributed Computing*, vol. 16, no. 2, pp. 108–120, 1992.

[64] REHG, J. M., KNOBE, K., RAMACHANDRAN, U., NIKHIL, R. S., and CHAUHAN, A., "Integrated Task and Data Parallel Support for Dynamic Applications," in *Proc. of 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, vol. 1511 of *Lecture Notes in Computer Science*, pp. 167–180, Pittsburgh, PA: Springer-Verlag, May 28–30 1998.

[65] REHG, J. M., KNOBE, K., RAMACHANDRAN, U., NIKHIL, R. S., and CHAUHAN, A., "Integrated Task and Data Parallel Support for Dynamic Applications," *Scientific Programming*, vol. 7, no. 3-4, pp. 289–302, 1999. Invited paper, selected from 1998 Workshop on Languages, Compilers, and Run-Time Systems (LCR98).

[66] REHG, J. M., LOUGHLIN, M., and WATERS, K., "Vision for a Smart Kiosk," in *Computer Vision and Pattern Recognition*, (San Juan, Puerto Rico), pp. 690–696, June 17–19 1997.

[67] REHG, J. M., RAMACHANDRAN, U., HALSTEAD, JR., R. H., JOERG, C., KONTOTHANASSIS, L., and NIKHIL, R. S., "Space-Time Memory: A Parallel Programming Abstraction for Dynamic Vision Applications," Tech. Rep. CRL 97/2, Digital Equipment Corp., Cambridge Research Laboratory, April 1997.

[68] RICHARD JONES AND RAFAEL LINS, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley, August 1996.

[69] RINARD, M. C., SCALES, D. J., and LAM, M. S., "Heterogeneous Parallel Programming in Jade," in *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, (Los Alamitos, CA, USA), pp. 245–256, IEEE Computer Society Press, 1992.

[70] RINARD, M. C., SCALES, D. J., and LAM, M. S., "Jade: A High-Level, Machine-Independent Language for Parallel Programming," *Computer*, vol. 26, no. 6, pp. 28–38, 1993.

[71] SAITO, H., STAVRAKOS, N. J., POLYCHRONOPOULOS, C. D., and NICOLAU, A., "The Design of the PROMIS Compiler – Towards Multi-Level Parallelization," *International Journal of Parallel Programming*, vol. 28, no. 2, pp. 195–212, 2000.

[72] SCHARSTEIN, D. and SZELISKI, R., "A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms," *International Journal of Computer Vision*, vol. 47, no. 1, pp. 7–42, 2002.

[73] SCHNEIDERMAN, H. and KANADE, T., "A statistical model for 3d object detection applied to faces and cars," in *IEEE Conference on Computer Vision and Pattern Recognition*, IEEE, June 2000.

[74] STEFFAN, J., COLOHAN, C., ZHAI, A., and MOWRY, T., "A Scalable Approach to Thread-Level Speculation," *Proceedings of the 27th annual international symposium on Computer architecture*, pp. 1–12, 2000.

[75] SUN MICROSYSTEMS, *JavaSpaces Service Specifications, version 1.2.1*. Palo Alto, California, USA: Sun Microsystems, April 2002.

[76] SUTTER, H. and LARUS, J., "Software and the Concurrency Revolution," *Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[77] VIOLA, P. and JONES, M., "Rapid Object Detection using a Boosted Cascade of Simple Features," *Computer Vision and Pattern Recognition*, vol. 01, p. 511, 2001.

[78] WATERS, K., REHG, J. M., LOUGHLIN, M., KANG, S. B., and TERZOPOULOS, D., "Visual Sensing of Humans for Active Public Interfaces," in *Computer Vision for Human-Machine Interaction* (CIPOLLA, R. and PENTLAND, A., eds.), Cambridge University Press, 1998.

[79] WILLIAM GROPP AND EWING LUSK AND ANTHONY SKJELLUM AND RAJEEV THAKUR, *Using MPI Portable Parallel Programming with the Message-Passing Interface Second Edition*. Cambridge, MA, USA: MIT Press, November 1999.

[80] WILLIAM GROPP AND STEVEN HUSS-LEDERMAN AND ANDREW LUMSDAINE AND EWIG LUSK AND BILL NITZBERG AND WILLIAM SAPHIR AND MARC SNIR, *MPI - The Complete Reference. Volume 2 - The MPI-2 Extensions*, vol. 2. Cambridge, MA, USA: MIT Press, September 1998.

[81] WILLIAM M. BEAN, *Greenberg's Guide to Gilbert Erector Sets: 1933-1962. Volume Two*. Greenberg Pub, February 1998.

[82] WILSON, P. R., "Uniprocessor garbage collection techniques, Yves Bekkers and Jacques Cohen (eds.)," in *International Workshop on Memory Management (IWMM 92)*, (St. Malo, France), pp. 1–42, September 1992.

[83] YANG, R. and POLLEFEYS, M., "A Versatile Stereo Implementation on Commodity Graphics Hardware," *Journal of Real-Time Imaging*, vol. 11, pp. 7–18, February 2005.

INDEX